



# Red Hat Data Grid 8.1

## Data Grid 開発者ガイド

Data Grid のカスタマイズ、設定、拡張





## 法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and other countries.

Java<sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

XFS<sup>®</sup> is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL<sup>®</sup> is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js<sup>®</sup> is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack<sup>®</sup> Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## 概要

Data Grid API、Data Grid 機能を使用するためにアプリケーションをコーディングする方法、および Data Grid のカスタマイズ方法について学びます。

## 目次

RED HAT DATA GRID .....	5
DATA GRID のドキュメント .....	6
DATA GRID のダウンロード .....	7
多様性を受け入れるオープンソースの強化 .....	8
<b>第1章 DATA GRID MAVEN リポジトリの設定 .....</b>	<b>9</b>
1.1. DATA GRID MAVEN リポジトリのダウンロード	9
1.2. RED HAT MAVEN リポジトリの追加	9
1.3. DATA GRID POM の設定	10
<b>第2章 キャッシュマネージャー .....</b>	<b>11</b>
2.1. キャッシュの取得	11
2.2. クラスタリング情報	12
2.3. メンバー情報	12
<b>第3章 DATA GRID キャッシュインターフェイス .....</b>	<b>13</b>
3.1. キャッシュ API	13
3.2. ADVANCEDCACHE API	14
3.3. リスナーおよび通知	14
3.4. ASYNCHRONOUS API	18
<b>第4章 キャッシュエンコーディングの設定 .....</b>	<b>20</b>
4.1. キャッシュエンコーディングとクライアントの相互運用性	20
4.2. CONFIGURING ENCODING FOR DATA GRID CACHES	22
4.3. PROTOBUF 形式でのデータの保存	23
4.4. テキストベースの形式でのデータの保存	24
4.5. マーシャリングされた JAVA オブジェクトの保存	25
4.6. アンマーシャリングされた JAVA オブジェクトの保存	25
4.7. データエンコーディング	26
4.8. トランスコーダーとデータ変換	30
<b>第5章 JAVA オブジェクトをマーシャルするための DATA GRID の設定 .....</b>	<b>33</b>
5.1. サポートされるタイプ	33
5.2. PROTOSTREAM を使用したユーザータイプのマーシャリング	33
5.3. 代替マーシャラー実装の設定	39
<b>第6章 クラスタ化されたロック .....</b>	<b>45</b>
6.1. ロック API	45
6.2. クラスタ化されたロックの使用	45
6.3. ロックの内部キャッシュの設定	47
<b>第7章 クラスタ化カウンター .....</b>	<b>49</b>
7.1. インストールおよび設定	49
7.2. COUNTERMANAGER インターフェイス	52
7.3. カウンター	52
7.4. 通知およびイベント	57
<b>第8章 ロックおよび同時実行 .....</b>	<b>59</b>
8.1. 実装の詳細のロック	59
8.2. データのバージョン管理	61
<b>第9章 DATA GRID CDI エクステンションの使用 .....</b>	<b>62</b>

9.1. CDI 依存関係	62
9.2. 組み込みキャッシュのインジェクト	62
9.3. リモートキャッシュの注入	64
9.4. JCACHE キャッシングアノテーション	65
9.5. キャッシュおよびキャッシュマネージャーイベントの受信	67
<b>第10章 DATA GRID トランザクション</b>	<b>68</b>
10.1. トランザクションの設定	68
10.2. 分離レベル	70
10.3. トランザクションのロック	71
10.4. スキューの書き込み	72
10.5. 例外への対処	73
10.6. 同期の登録	73
10.7. バッチ処理	73
10.8. トランザクションリカバリー	74
<b>第11章 インデックス化および検索</b>	<b>79</b>
11.1. 概要	79
11.2. エントリー値のインデックス化	79
11.3. 検索	82
11.4. 埋め込み検索	89
11.5. リモート検索	93
11.6. 継続的なクエリー	98
11.7. 統計	101
11.8. パフォーマンスチューニング	102
<b>第12章 GRID でのコードの実行</b>	<b>104</b>
12.1. クラスターエグゼキューター	104
<b>第13章 ストリーム</b>	<b>108</b>
13.1. 一般的なストリーム操作	108
13.2. キーのフィルタリング	108
13.3. セグメントベースのフィルタリング	108
13.4. ローカル/無効化	109
13.5. 例	109
13.6. 配布/複製/散在	109
13.7. 並列計算	112
13.8. タスクのタイムアウト	112
13.9. 注入	113
13.10. 分散ストリームの実行	113
13.11. キーベースの再ハッシュ対応 OPERATOR	114
13.12. 中間オペレーションの例外	115
13.13. 例	115
<b>第14章 JCACHE (JSR-107) API</b>	<b>118</b>
14.1. 組み込みキャッシュの作成	118
14.2. リモートキャッシュの作成	119
14.3. データの保管および取得	120
14.4. JAVA.UTIL.CONCURRENT.CONCURRENTMAP と JAVAX.CACHE.CACHE APIS の比較	120
14.5. JCACHE インスタンスのクラスタリング	122
<b>第15章 マルチマップキャッシュ</b>	<b>123</b>
15.1. インストールと設定	123
15.2. MULTIMAPCACHE API	123
15.3. マルチマップキャッシュの作成	125

---

15.4. 制限事項	125
<b>第16章 RED HAT JBOSS EAP のデータグリッドモジュール</b> .....	<b>126</b>
16.1. DATA GRID モジュールのインストール	126
16.2. DATA GRID モジュールを使用するためのアプリケーションの設定	126
<b>第17章 RED HAT JBOSS WEB SERVER から RED HAT DATA GRID への HTTP セッションの外部化</b> .....	<b>128</b>
17.1. TOMCAT セッションクライアントのインストール	128
17.2. セッションマネージャーの設定	128
<b>第18章 カスタムインターセプター</b> .....	<b>131</b>
18.1. カスタムインターセプターの宣言的追加	131
18.2. プログラムによるカスタムインターセプターの追加	131
18.3. カスタムインターセプターの設計	131





---

# RED HAT DATA GRID

Data Grid は、高性能の分散型インメモリーデータストアです。

## スキーマレスデータ構造

さまざまなオブジェクトをキーと値のペアとして格納する柔軟性があります。

## グリッドベースのデータストレージ

クラスター間でデータを分散および複製するように設計されています。

## エラスティックスケールリング

サービスを中断することなく、ノードの数を動的に調整して要件を満たします。

## データの相互運用性

さまざまなエンドポイントからグリッド内のデータを保存、取得、およびクエリーします。

## DATA GRID のドキュメント

Data Grid のドキュメントは、Red Hat カスタマーポータルで入手できます。

- [Data Grid 8.1 ドキュメント](#)
- [Data Grid 8.1 コンポーネントの詳細](#)
- [Data Grid 8.1 でサポートされる設定](#)
- [Data Grid 8 機能のサポート](#)
- [Data Grid で非推奨の機能](#)

## DATA GRID のダウンロード

Red Hat カスタマーポータルで [Data Grid Software Downloads](#) にアクセスします。



### 注記

Data Grid ソフトウェアにアクセスしてダウンロードするには、Red Hat アカウントが必要です。

## 多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#) をご覧ください。

# 第1章 DATA GRID MAVEN リポジトリの設定

Data Grid Java ディストリビューションは Maven から入手できます。

顧客ポータルから Data Grid Maven リポジトリをダウンロードするか、パブリック Red Hat Enterprise Maven リポジトリから Data Grid 依存関係をプルできます。

## 1.1. DATA GRID MAVEN リポジトリのダウンロード

パブリック Red Hat Enterprise Maven リポジトリを使用しない場合は、ローカルファイルシステム、Apache HTTP サーバー、または Maven リポジトリマネージャーに Data Grid Maven リポジトリをダウンロードし、インストールします。

### 手順

1. Red Hat カスタマーポータルにログインします。
2. [Software Downloads for Data Grid](#) に移動します。
3. Red Hat Data Grid 8.1 Maven リポジトリをダウンロードします。
4. アーカイブされた Maven リポジトリをローカルファイルシステムにデプロイメントします。
5. **README.md** ファイルを開き、適切なインストール手順に従います。

## 1.2. RED HAT MAVEN リポジトリの追加

Red Hat GA リポジトリを Maven ビルド環境に組み込み、Data Grid アーティファクトおよび依存関係を取得します。

### 手順

- Red Hat GA リポジトリを Maven 設定ファイル (通常は `~/.m2/settings.xml`) に追加するか、プロジェクトの `pom.xml` ファイルに直接追加します。

```
<repositories>
  <repository>
    <id>redhat-ga-repository</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga-repository</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
</pluginRepositories>
```

### 参照資料

- [Red Hat Enterprise Maven Repository](#)

## 1.3. DATA GRID POM の設定

Maven は、プロジェクトオブジェクトモデル (POM) ファイルと呼ばれる設定ファイルを使用して、プロジェクトを定義し、ビルドを管理します。POM ファイルは XML 形式であり、モジュールとコンポーネントの依存関係、ビルドの順序、および結果となるプロジェクトのパッケージ化と出力のターゲットを記述します。

### 手順

1. プロジェクト **pom.xml** を開いて編集します。
2. 正しい Data Grid バージョンで **version.infinispan** プロパティを定義します。
3. **dependencyManagement** セクションに **infinispan-bom** を含めます。  
BOM(Bill of Materials) は、依存関係バージョンを制御します。これにより、バージョンの競合が回避され、プロジェクトに依存関係として追加する Data Grid アーティファクトごとにバージョンを設定する必要がなくなります。
4. **pom.xml** を保存して閉じます。

以下の例は、Data Grid のバージョンと BOM を示しています。

```
<properties>
  <version.infinispan>11.0.9.Final-redhat-00001</version.infinispan>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

### 次のステップ

必要に応じて、Data Grid アーティファクトを依存関係として **pom.xml** に追加します。

## 第2章 キャッシュマネージャー

Data Grid への主なエントリーポイントは **CacheManager** インターフェイスであり、以下を行うことができます。

- キャッシュを設定および取得します。
- クラスタ化された Data Grid ノードを管理および監視します。
- クラスタ全体でコードを実行します。

Data Grid をアプリケーションに埋め込む場合は、**EmbeddedCacheManager** を使用します。Data Grid をリモートサーバーとして実行する場合は、**RemoteCacheManager** を使用します。

キャッシュマネージャーはヘビーウェイトオブジェクトであるため、ほとんどの場合、JVM ごとに1つの **CacheManager** インスタンスのみをインスタンス化する必要があります。

```
EmbeddedCacheManager manager = new DefaultCacheManager(); ❶
```

❶ キャッシュのないローカルのクラスタ化されていないキャッシュマネージャーを起動します。

キャッシュマネージャーにはライフサイクルがあり、デフォルトのコンストラクターは **start()** メソッドも呼び出します。コンストラクターのオーバーロードされたバージョンが利用可能ですが、それらは **CacheManager** を開始しません。ただし、キャッシュを作成する前に必ず **CacheManager** を起動する必要があります。

同様に、実行中の **CacheManager** が不要になった際に **stop()** を呼び出して、リソースを解放する必要があります。これにより、キャッシュマネージャーは制御するキャッシュを安全に停止することもできます。

### 2.1. キャッシュの取得

**CacheManager** を設定した後、キャッシュを取得および制御できます。

以下のように、キャッシュを取得するために **getCache(String)** メソッドを呼び出します。

```
Cache<String, String> myCache = manager.getCache("myCache");
```

上記の操作は、**myCache** という名前のキャッシュがまだ存在しない場合は作成し、それを返します。

**getCache()** メソッドを使用すると、メソッドを呼び出すノードにのみキャッシュが作成されます。つまり、クラスタ全体の各ノードで呼び出す必要のあるローカル操作を実行します。通常、複数のノードにまたがってデプロイされたアプリケーションは、初期化中にキャッシュを取得して、キャッシュが対称であり、各ノードに存在することを確認します。

**createCache()** メソッドを呼び出して、以下のようにクラスタ全体でキャッシュを動的に作成します。

```
Cache<String, String> myCache = manager.administration().createCache("myCache",  
"myTemplate");
```

上記の操作では、後でクラスタに参加するすべてのノードにキャッシュが自動的に作成されます。

**createCache()** メソッドを使用して作成するキャッシュは、デフォルトでは一時的です。クラスター全体がシャットダウンした場合、再起動時にキャッシュが自動的に再作成されることはありません。

PERMANENT フラグを使用して、以下のようにキャッシュが再起動後も存続できるようにします。

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

PERMANENT フラグを有効にするには、グローバルの状態を有効にし、設定ストレージプロバイダーを設定する必要があります。

設定ストレージプロバイダーの詳細は、[GlobalStateConfigurationBuilder#configurationStorage\(\)](#) を参照してください。

## 2.2. クラスタリング情報

**EmbeddedCacheManager** には、クラスターの動作に関する情報を提供するためのメソッドが多数あります。以下のメソッドは、クラスター環境で使用される場合 (Transport が設定されている場合) にのみ意味があります。

### 2.3. メンバー情報

クラスターを使用している場合、クラスターの所有者が誰であるかなど、クラスターのメンバーシップに関する情報を見つけれることが非常に重要となります。

#### [getMembers\(\)](#)

`getMembers()` メソッドは、現在のクラスター内のすべてのノードを返します。

#### [getCoordinator\(\)](#)

`getCoordinator()` メソッドは、どのメンバーがクラスターのコーディネーターであるかを指示します。ほとんどの場合、コーディネーターが誰であるかを気にする必要はありません。[isCoordinator\(\)](#) メソッドを直接使用して、ローカルノードがコーディネーターであるかどうかを確認することもできます。



## 第3章 DATA GRID キャッシュインターフェイス

Data Grid は、JDK の `ConcurrentMap` インターフェイスによって公開されるアトミックメカニズムを含む、エントリーを追加、取得、および削除するための簡単なメソッドを公開する `Cache` インターフェイスを提供します。使用されるキャッシュモードに基づいて、これらのメソッドを呼び出すと、リモートノードにエントリーを複製したり、リモートノードからエントリーを検索することやキャッシュストアからエントリーを検索することなど、数多くのことが発生します。

### 3.1. キャッシュ API

単純な使用の場合、`Cache` API の使用は `JDK Map` API の使用と違いがないはずです。したがって、マップに基づく単純なインメモリーキャッシュから Data Grid のキャッシュへの移行は簡単になります。

#### 3.1.1. 特定のマップメソッドのパフォーマンスに関する懸念

`size()`、`values()`、`keySet()`、および `entrySet()` など、マップで公開される特定のメソッドは、Data Grid と使用すると特定のパフォーマンスに影響します。`keySet` の特定のメソッドである `values` および `entrySet` を使用できます。詳細については、Javadoc を参照してください。

これらの操作をグローバルに実行しようとする、パフォーマンスに大きな影響を及ぼし、スケラビリティのボトルネックにもなります。そのため、これらの方法は情報またはデバッグの目的でのみ使用してください。

`withFlags()` メソッドで特定のフラグを使用すると、これらの問題の一部を軽減できます。詳細は、各メソッドのドキュメントを参照してください

#### 3.1.2. Mortal および Immortal データ

単にエントリーを格納するだけでなく、Data Grid のキャッシュ API を使用すると、期限付き情報をデータに添付できます。たとえば、単に `put(key, value)` を使用すると、`immortal` エントリーが作成されます。このエントリーは削除されるまで (またはメモリー不足にならないようにメモリーからエビクトされるまで)、いつまでもキャッシュに存在します。ただし、`put(key, value, lifespan, timeunit)` を使用してキャッシュにデータを配置すると、`mortal` エントリーが作成されます。これは固定のライフスパンのあるエントリーで、そのライフスパンの後に期限切れになります。

Data Grid は、`lifespan` の他に、有効期限を決定する追加のメトリックとして `maxIdle` もサポートします。`lifespans` または `maxIdles` の任意の組み合わせを使用できます。

#### 3.1.3. putForExternalRead 操作

Data Grid の `Cache` クラスには、`putForExternalRead` と呼ばれる異なる 'put' 操作が含まれます。この操作は、他の場所で保持されるデータの一時キャッシュとして Data Grid が使用される場合に特に便利です。読み取りが非常に多い場合、キャッシュは単に最適化のために行われ、妨害するものではないため、キャッシュの競合によって実際のトランザクションが遅延してはなりません。

これを実現するため、キーがキャッシュ内に存在しない場合にのみ動作する `put` 呼び出しとして `putForExternalRead()` が動作し、別のスレッドが同じキーを同時に格納しようとする、通知なしに即座に失敗します。この特定のシナリオでは、データのキャッシュはシステムを最適化する方法であり、キャッシュの失敗が進行中のトランザクションに影響することが望まれないため、失敗は異なる理由です。成功したかどうかに関係なく、`putForExternalRead ()` は高速操作と見なされます。ロックを待たずにすぐに呼び出し元に返すことはありません。

この操作の使用方法を理解するために、基本的な例を見てみましょう。PersonId によって入力される

Person インスタンスのキャッシュを想像してください。このデータは個別のデータストアから入力されます。以下のコードは、この例のコンテキスト内で `putForExternalRead` を使用する最も一般的なパターンを示しています。

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
    // The person is not cached yet, so query the data store with the id
    Person person = datastore.lookup(id);

    // Cache the person along with the id so that future requests can
    // retrieve it from memory rather than going to the data store
    cache.putForExternalRead(id, person);
} else {
    // The person was found in the cache, so return it to the application
    return cachedPerson;
}
```

`putForExternalRead` は、アプリケーションの実行元 (Person のアドレスを変更するトランザクションからなど) となる新しい Person インスタンスでキャッシュを更新するメカニズムとして使用しないでください。キャッシュされた値を更新する場合は、標準の `put` 操作を使用してください。使用しないと、破損したデータをキャッシュする可能性が高くなります。

## 3.2. ADVANCEDCACHE API

簡単なキャッシュインターフェイスの他に、Data Grid はエクステンション作成者向けに `AdvancedCache` インターフェイスを提供します。`AdvancedCache` は、特定の内部コンポーネントにアクセスし、フラグを適用して特定のキャッシュメソッドのデフォルト動作を変更する機能を提供します。次のコードスニペットは、`AdvancedCache` を取得する方法を示しています。

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

### 3.2.1. フラグ

フラグは通常のキャッシュメソッドに適用され、特定のメソッドの動作を変更します。利用可能なフラグの一覧と、その効果については、[Flag](#) 列挙を参照してください。フラグは、`AdvancedCache.withFlags()` を使用して適用されます。このビルダーメソッドを使用して、キャッシュ呼び出しに任意の数のフラグを適用できます。次に例を示します。

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
    .withFlags(Flag.FORCE_SYNCHRONOUS)
    .put("hello", "world");
```

## 3.3. リスナーおよび通知

Data Grid はリスナー API を提供し、クライアントはイベントが発生したときに登録して通知を受け取ることができます。このアノテーション駆動型 API は、キャッシュレベルイベントとキャッシュマネージャーレベルイベントの2つの異なるレベルに適用されます。

イベントは、リスナーにディスパッチされる通知をトリガーします。リスナーは `@Listener` アノテーションが付けられ、`Listenable` インターフェイスで定義されたメソッドを使用して登録された単純な POJO です。



### 注記

Cache と CacheManager はどちらも `Listenable` を実装しています。つまり、リスナーをキャッシュまたはキャッシュマネージャーのいずれかにアタッチして、キャッシュレベルまたはキャッシュマネージャーレベルのいずれかの通知を受信できます。

たとえば、次のクラスは、新しいエントリーがキャッシュに追加されるたびに、ブロックしない方法で、一部の情報を出力するようにリスナーを定義します。

```
@Listener
public class PrintWhenAdded {
    Queue<CacheEntryCreatedEvent> events = new ConcurrentLinkedQueue<>();

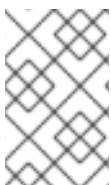
    @CacheEntryCreated
    public CompletionStage<Void> print(CacheEntryCreatedEvent event) {
        events.add(event);
        return null;
    }
}
```

より包括的な例は [Javadocs for @Listener](#) を参照してください。

### 3.3.1. キャッシュレベルの通知

キャッシュレベルのイベントはキャッシュごとに発生し、デフォルトでは、イベントが発生したノードでのみ発生します。分散キャッシュでは、これらのイベントは影響を受けるデータの所有者に対してのみ発生することに注意してください。キャッシュレベルのイベントの例としては、エントリーの追加、削除、変更などがあります。これらのイベントは、特定のキャッシュに登録されているリスナーへの通知をトリガーします。

すべてのキャッシュレベルの通知とそれぞれのメソッドレベルのアノテーションの包括的なリストについては、[org.infinispan.notifications.cachelistener.annotation](#) パッケージの [Javadocs](#) を参照してください。



### 注記

Data Grid で使用可能なキャッシュレベルの通知のリストについては [org.infinispan.notifications.cachelistener.annotation](#) パッケージの [Javadocs](#) を参照してください。

#### 3.3.1.1. クラスタリスナー

単一ノードでキャッシュイベントをリッスンすることが望ましい場合は、クラスタリスナーを使用する必要があります。

そのために必要なのは、リスナーがクラスター化されているというアノテーションを付けるよう設定することだけです。

```
@Listener (clustered = true)
public class MyClusterListener { .... }
```

クラスター化されていないリスナーからのクラスターリスナーには、いくつかの制限があります。

1. クラスターリスナー  
は、**@CacheEntryModified**、**@CacheEntryCreated**、**@CacheEntryRemoved**、および**@CacheEntryExpired** イベントのみをリスンできます。これは、他のタイプのイベントは、このリスナーに対してリスンされないことを意味することに注意してください。
2. ポストイベントのみがクラスターリスナーに送信され、プレイベントは無視されます。

### 3.3.1.2. イベントのフィルタリングおよび変換

リスナーがインストールされているノードで適用可能なすべてのイベントがリスナーに発生します。**KeyFilter**(キーのフィルタリングのみを許可)または**CacheEventFilter**(キー、古い値、古いメタデータ、新しい値、新しいメタデータ、コマンドの再実行の有無、イベントがイベント(isPre など)の前であるか、およびコマンドタイプのフィルターに使用)を使用して、どのイベントが発生したかを動的にフィルターできます。

この例で、イベントがキー **Only Me** のエントリーを変更したときにイベントのみを発生させる単純な**KeyFilter**を示しています。

```
public class SpecificKeyFilter implements KeyFilter<String> {
    private final String keyToAccept;

    public SpecificKeyFilter(String keyToAccept) {
        if (keyToAccept == null) {
            throw new NullPointerException();
        }
        this.keyToAccept = keyToAccept;
    }

    public boolean accept(String key) {
        return keyToAccept.equals(key);
    }
}

...
cache.addListener(listener, new SpecificKeyFilter("Only Me"));
...
```

これは、より効率的な方法で受信するイベントを制限したい場合に便利です。

また、イベントが発生する前に値を別の値に変換できるようにする**CacheEventConverter**もあります。これは、値の変換を行うコードをモジュール化するのに適しています。



## 注記

上記のフィルターとコンバーターは、クラスターリスナーと組み合わせて使用すると特に効果的です。これは、イベントがリスンされているノードではなく、イベントが発生したノードでフィルタリングと変換が行われるためです。これにより、クラスター全体でイベントを複製する必要がない(フィルター)、またはペイロードを減らす(コンバーター)という利点があります。

### 3.3.1.3. 初期状態のイベント

リスナーがインストールされると、完全にインストールされた後にのみイベントが通知されます。

リスナーの初回登録時にキャッシュコンテンツの現在の状態を取得することが望ましい場合があります。この場合、キャッシュ内の各要素の **@CacheEntryCreated** タイプのイベントが生成されます。この最初のフェーズで追加で生成されたイベントは、適切なイベントが発生するまでキューに置かれます。



## 注記

現時点では、これはクラスター化されたリスナーに対してのみ機能します。 [ISPN-4608](#) では、クラスター化されていないリスナーへの追加を説明しています。

### 3.3.1.4. 重複イベント

トランザクションではないキャッシュで、重複したイベントを受け取ることが可能です。これは、put などの書き込み操作の実行中に、キーのプライマリ所有者がダウンした場合に可能になります。

Data Grid は、指定のキーの新規プライマリ所有者に put 操作を自動的に送信することで、put 操作を内部で修正しますが、最初に書き込みがバックアップに複製されたかどうかについては保証はありません。そのため、**CacheEntryCreatedEvent**、**CacheEntryModifiedEvent**、および **CacheEntryRemovedEvent** の書き込みイベントの1つ以上が、1つの操作で送信される可能性があります。

複数のイベントが生成された場合、Data Grid は再試行コマンドによって生成されたイベントをマークし、変更の表示に注意を払いなくても、このイベントが発生したタイミングを把握できるようにします。

```
@Listener
public class MyRetryListener {
    @CacheEntryModified
    public void entryModified(CacheEntryModifiedEvent event) {
        if (event.isCommandRetried()) {
            // Do something
        }
    }
}
```

また、**CacheEventFilter** または **CacheEventConverter** を使用する場合、**EventType**には、再試行によりイベントが生成されたかどうかを確認するために、メソッド **isRetry** が含まれます。

### 3.3.2. キャッシュマネージャーレベルの通知

キャッシュマネージャーレベルのイベントは、キャッシュマネージャーで行われます。これらはグローバルでクラスター全体でもありますが、単一のキャッシュマネージャーによって作成されたすべてのキャッシュに影響するイベントが関係します。キャッシュマネージャーレベルのイベントの例として、

クラスターに参加または退出するノード、または開始または停止するキャッシュがあります。

キャッシュマネージャーレベルのすべての通知とそれぞれのメソッドレベルのアノテーションの包括的なリストは、[org.infinispan.notifications.cachemanagerlistener.annotation package](https://docs.redhat.com/en/documentation/red_hat_data_grid/8.1/html-developer-guide/notifications_cachemanagerlistener_annotation_package) を参照してください。

### 3.3.3. イベントの同期

デフォルトでは、すべての非同期通知は通知スレッドプールにディスパッチされます。同期通知は、リスナーメソッドが完了するか (スレッドがブロックする原因となる)、または `CompletionStage` が完了するまで、操作の続行を遅らせます。または、リスナーに**非同期**としてアノテーションを付けることもできます。この場合、操作は即座に継続され、通知は通知スレッドプールで非同期に完了します。これには、以下のようにリスナーにアノテーションを付けます。

非同期リスナー

```
@Listener (sync = false)
public class MyAsyncListener {
    @CacheEntryCreated
    void listen(CacheEntryCreatedEvent event) {}
}
```

同期リスナーのブロック

```
@Listener
public class MySyncListener {
    @CacheEntryCreated
    void listen(CacheEntryCreatedEvent event) {}
}
```

ノンブロッキングリスナー

```
@Listener
public class MyNonBlockingListener {
    @CacheEntryCreated
    CompletionStage<Void> listen(CacheEntryCreatedEvent event) {}
}
```

#### 3.3.3.1. 非同期スレッドプール

このような非同期通知のディスパッチに使用されるスレッドプールを調整するには、設定ファイルの `<listener-executor />` XML 要素を使用します。

## 3.4. ASYNCHRONOUS API

`Cache.put()`、`Cache.remove()` などの同期 API メソッドの他に、Data Grid には非同期のノンブロッキング API も含まれ、同じ結果をノンブロッキング方式で達成できます。

これらのメソッドの名前は、ブロックメソッドと同様の名前が付けられ、"Async"が追加されます。例: `Cache.putAsync()`、`Cache.removeAsync()` など。これらの非同期のメソッドは、操作の実際の結果が含まれる `CompletableFuture` を返します。



たとえば、`Cache<String, String>` としてパラメーター化されたキャッシュでは、`Cache.put(String key, String value)` は `String` を返し、`Cache.putAsync(String key, String value)` は `CompletableFuture<String>` を返します。

### 3.4.1. このような API を使用する理由

ノンブロッキング API は、通信の失敗や例外を処理する機能を備えており、同期通信の保証をすべて提供するという点で強力なもので、呼び出しが完了するまでブロックする必要がありません。これにより、システムで並列処理をより有効に活用できます。以下に例を示します。

```
Set<CompletableFuture<?>> futures = new HashSet<>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (CompletableFuture<?> f: futures) f.get();
```

### 3.4.2. 実際に非同期で発生するプロセス

Data Grid には、通常書き込み操作の重要なパスにあると見なされる 4 つの項目があります。これらの項目をコスト順に示します。

- ネットワークコール
- マーシャリング
- キャッシュストアへの書き込み (オプション)
- ロック

非同期メソッドを使用すると、ネットワーク呼び出しとマーシャリングがクリティカルパスから除外されます。ただし、さまざまな技術的な理由により、キャッシュストアへの書き込みとロックの取得は、呼び出し元のスレッドで引き続き発生します。

## 第4章 キャッシュエンコーディングの設定

Data Grid は、キャッシュとの間で読み取りおよび書き込みを行うときにオンザフライで変換できる特定の形式でデータを保存します。キーと値の MediaType を指定して、ストレージ形式を設定します。これはデータの形式を記述します。

Data Grid は、異なるストレージ形式間でデータを変換して、異なるクライアントプロトコル間の相互運用性を処理したり、カスタムコードを使用してデータを処理したりすることもできます。

### 4.1. キャッシュエンコーディングとクライアントの相互運用性

データに使用するエンコーディングは、クライアントの相互運用性と Data Grid Search などの機能に影響します。

表4.1 Protobuf 形式

Protobuf 形式でデータを保存して、以下と使用します。	
Data Grid コンソール	はい
REST クライアント	はい
Java Hot Rod クライアント	はい
Java 以外の Hot Rod クライアント	はい
Data Grid Search	はい
カスタム Java オブジェクト	はい

表4.2 テキストベースの形式

テキストベースの形式でデータを保存して、以下と使用します。	
Data Grid コンソール	はい
REST クライアント	はい
Java Hot Rod クライアント	はい
Java 以外の Hot Rod クライアント	はい
Data Grid Search	いいえ
カスタム Java オブジェクト	いいえ

表4.3 マーシャリングされた Java オブジェクト



マーシャリングされた Java オブジェクトは以下と互換性があります。

Data Grid コンソール	いいえ
REST クライアント	はい
Java Hot Rod クライアント	はい
Java 以外の Hot Rod クライアント	いいえ
Data Grid Search	いいえ

表4.4 アンマーシャリングされた Java オブジェクト

POJO (Plain Old Java Object) は推奨されませんが、以下と互換性があります。

Data Grid コンソール	いいえ
REST クライアント	はい
Java Hot Rod クライアント	はい
Java 以外の Hot Rod クライアント	いいえ
Data Grid Search	必要。ただし、POJO で検索し、クラスを Data Grid Server で使用できるようにするには、エンティティーにアノテーションを付ける必要があります。
カスタム Java オブジェクト	はい

## 参照

- [Configuring Encoding for Data Grid Caches](#)
- [Transcoders and Data Conversion](#)
- [Data Encoding](#)

### 4.1.1. Memcached クライアントのキャッシュエンコーディングの設定

Data Grid Server は、デフォルトで Memcached エンドポイントを無効にします。Memcached エンドポイントを有効にする場合は、Memcached クライアントに適したエンコーディングでキャッシュを設定する必要があります。



## 重要

Memcached エンドポイントは認証をサポートしません。セキュリティ上の理由から、Memcached クライアント専用のキャッシュを使用する必要があります。Memcached クライアントと同じデータセットで対話するために REST または Hot Rod クライアントを使用しないでください。

## 手順

1. キーに **text/plain** を使用するようにキャッシュエンコーディングを設定します。
2. **application/x-java-object** 以外の適切な MediaType を値に指定します。  
Memcached クライアントは、鍵を **text/plain** としてのみ処理できます。値は、Data Grid が **byte[]** として保存する MediaType にすることができます。これは Protobuf、マーシャリングされた Java オブジェクト、またはテキストベースの形式になります。

```
<encoding>
  <key media-type="text/plain"/>
  <value media-type="application/x-protostream"/>
</encoding>
```

## ヒント

Memcached エンドポイントには、値のエンコーディングを変換する **client-encoding** 属性が含まれません。

たとえば、前述の設定の例にあるように、Protobuf としてエンコードされた値を保存します。Memcached クライアントで値を JSON として読み書きできるようにする場合は、次の設定を使用できます。

```
<memcached-connector cache="memcachedCache" client-encoding="application/json">
```

## 参照

[Red Hat ナレッジベースの記事:Data Grid の Memcached エンドポイント](#)

## 4.2. CONFIGURING ENCODING FOR DATA GRID CACHES

キャッシュとの間で書き込みおよび読み取りを行うときに Data Grid がデータをエンコードするために使用する MediaType を定義します。

## ヒント

MediaType を定義するときに、Data Grid にデータの形式を指定します。

Data Grid Console、Hot Rod クライアント、および REST クライアントを同じように使用する場合は、**application/x-protostream** を指定して、DataGrid がデータを Protobuf 形式でエンコードするようにします。

## 手順

- Data Grid キャッシュ設定で、キーと値の MediaType を指定します。
  - 宣言的に行う **encoding** 属性を設定します。

- プログラムで行う:`encoding()` メソッドを使用します。

#### 宣言的に行う場合の例

- キーと値に同じエンコーディングを使用します。

```
<local-cache>
  <encoding media-type="application/x-protostream"/>
</local-cache>
```

- キーと値に異なるエンコーディングを使用します。

```
<cache>
  <encoding>
    <key media-type="application/x-java-object"/>
    <value media-type="application/xml; charset=UTF-8"/>
  </encoding>
</cache>
```

#### プログラムで行う場合の例

- キーと値に同じエンコーディングを使用します。

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
  .mediaType("application/x-protostream")
  .build();
```

- キーと値に異なるエンコーディングを使用します。

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg.encoding().key().mediaType("text/plain");
cfg.encoding().value().mediaType("application/json");
```

### 4.3. PROTOBUF 形式でのデータの保存

Protobuf でエンコードされたエントリーとしてデータをキャッシュに保存すると、プラットフォームに依存しない設定が提供され、任意のクライアントからキャッシュ操作を実行できます。



#### 注記

Data Grid Search のインデックスを設定すると、Data Grid は **application/x-protostream** メディアタイプでキーと値を自動的に保存します。

#### 手順

1. **application/x-protostream** を、以下のようにキーと値の MediaType として指定します。

```
<distributed-cache name="mycache">
```

```
<encoding>
  <key media-type="application/x-protostream"/>
  <value media-type="application/x-protostream"/>
</encoding>
</distributed-cache>
```

2. クライアントを設定します。

Hot Rod クライアントは、エンティティおよびクライアントマーシャラーを記述する Protocol Buffers スキーマ定義を登録する必要があります。

Data Grid は **application/x-protostream** と **application/json** の間で変換されるため、REST クライアントは次のヘッダーを送信するだけで JSON 形式のデータを読み書きできます。

- **Accept: application/json**(読み取り操作の場合)
- **Content-Type: application/json**(書き込み操作の場合)

## 参照

- [Storage Formats and Client Interoperability](#)
- [Using the ProtoStream Marshaller](#)
- [Marshalling Custom Java Objects with ProtoStream](#)

## 4.4. テキストベースの形式でのデータの保存

**text/plain**、**application/json**、または **application/xml** などのテキストベースの形式でデータを保存するように Data Grid を設定します。

### 手順

1. キーと値の MediaType としてテキストベースのストレージ形式を指定します。
2. オプションで **UTF-8** などの文字セットを指定します。  
以下の例では、**text/plain; charset=UTF-8** 形式でエントリーを格納するように Data Grid を設定します。

```
<cache>
  <encoding>
    <key media-type="text/plain; charset=UTF-8"/>
    <value media-type="text/plain; charset=UTF-8"/>
  </encoding>
</cache>
```

3. クライアントを設定します。

Hot Rod クライアントは **org.infinispan.commons.marshall.StringMarshaller** を使用して、プレーンテキスト、JSON、XML、またはその他のテキストベースの形式を処理できます。

ProtoStream マーシャラーでテキストベースの形式を使用することもできます。ProtoStream は、シリアル化 コンテキストを作成したり Protobuf スキーマ (**.proto** ファイル) を登録したりする必要なく、**String** および **byte[]** タイプをネイティブに処理できます。

REST クライアントは、リクエストとともに正しいヘッダーを送信する必要があります。

- **Accept: text/plain; charset=UTF-8**(読み取り操作の場合)
- **Content-Type: text/plain; charset=UTF-8** (書き込み操作の場合)

## 4.5. マーシャリングされた JAVA オブジェクトの保存

Java Hot Rod クライアントは、エンティティーを表す Java オブジェクトを処理し、マーシャリングを実行し、オブジェクトを **byte[]** アレイにシリアライズおよびデシリアライズできます。C++、C#、および Javascript Hot Rod クライアントは、それぞれの言語のオブジェクトを処理することもできます。

エントリーをマーシャリングされた Java オブジェクトとして保存する場合は、マーシャリングされたストレージの MediaType でキャッシュを設定する必要があります。

### 手順

1. マーシャラー実装に一致する MediaType を指定します。
  - ProtoStream マーシャラー:MediaType を **application/x-protostream** として設定します。
  - JBoss marshalling:MediaType を **application/x-jboss-marshalling** として設定します。
  - Java シリアライゼーション:MediaType を **application/x-java-serialized-object** として設定します。
2. クライアントを設定します。

REST クライアントはテキスト形式の処理に最も適しているため、キーには **java.lang.String** などのプリミティブを使用する必要があります。それ以外の場合、REST クライアントは、サポートされるバイナリーエンコーディングを使用してキーを **bytes[]** として処理する必要があります。

REST クライアントは、XML または JSON 形式でキャッシュエントリーの値を読み取ることができます。

### 等価性に関する考慮事項

データをバイナリー形式で保存する場合、Data Grid はキーと値に **WrappedBytes** インターフェイスを使用します。このラッパークラスは、オンデマンドでシリアライズおよびデシリアライズを透過的に処理し、内部的には、ラップされているオブジェクト自体への参照、またはオブジェクトのシリアライズされたバイトアレイ表現を持つ場合があります。これは等価性の動作に影響します。これは、キーに **equals()** メソッドを実装する場合に注意することが重要です。

ラッパークラスの **equals()** メソッドは、比較対象の両方のインスタンスが比較時にシリアライズされた形式かデシリアライズされた形式かによって、バイナリー表現 (バイト配列) を比較するか、ラップされたオブジェクトインスタンスの **equals()** メソッドに委任します。比較される2つのインスタンスの形式が異なる場合、1つのインスタンスはシリアライズまたはデシリアライズされます。

### 参照

- [Marshalling Java Objects](#)
- [org.infinispan.commons.marshall.WrappedBytes](#)

## 4.6. アンマーシャリングされた JAVA オブジェクトの保存

データをバイナリー形式で保存する代わりに、デシリアライズされた Plain Old Java Objects(POJO) として保存できます。

バイナリー形式の代わりに POJO を保存することは推奨されていません。これは、Data Grid がクライアントの読み取り操作でデータをシリアライズし、書き込み操作でデータをデシリアライズする必要があるためです。クライアントとカスタムコードの相互運用性を処理するには、データをオンデマンドで変換する必要があります。

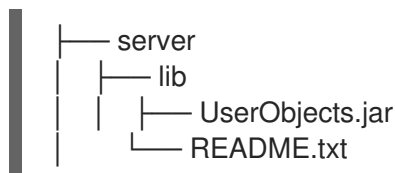
## 手順

1. 以下のように、キーと値の MediaType として **application/x-java-object** を指定します。

```
<distributed-cache name="my-cache">
  <encoding>
    <key media-type="application/x-java-object"/>
    <value media-type="application/x-java-object"/>
  </encoding>
</distributed-cache>
```

2. すべてのカスタムオブジェクトのクラスファイルを Data Grid サーバーのクラスパスに配置します。

**server/lib** ディレクトリーにマーシャラー実装用のカスタムクラスやサービスプロバイダーを含む JAR ファイルを追加します。



3. クライアントを設定します。

Hot Rod クライアントに必要な変更はありません。唯一の要件は、クライアントで使用されるマーシャラーが **server/lib** ディレクトリーで使用可能であり、Data Grid がオブジェクトをデシリアライズできるようにすることです。



### 注記

ProtoStream および Java Serialization マーシャラーはサーバー上ですでに利用できません。

REST クライアントは、JSON または XML を使用して、Data Grid が Java オブジェクトとの間で変換できるようにする必要があります。

## 参照

- [Converting Data on Demand](#)
- [Using the ProtoStream Marshaller](#)
- [Marshalling Custom Java Objects with ProtoStream](#)

## 4.7. データエンコーディング

エンコーディングは、データを保存する前、およびストレージから読み戻すときに、Data Grid キャッシュによって実行されるデータ変換操作です。

#### 4.7.1. 概要

エンコーディングを使用すると、API 呼び出し中に特定のデータ形式 (マップ、リスナー、ストリームなど) を処理できますが、効果的に保存される形式は異なります。

データ変換は、`org.infinispan.commons.dataconversion.Encoder` のインスタンスによって処理されます。

```
public interface Encoder {

    /**
     * Convert data in the read/write format to the storage format.
     *
     * @param content data to be converted, never null.
     * @return Object in the storage format.
     */
    Object toStorage(Object content);

    /**
     * Convert from storage format to the read/write format.
     *
     * @param content data as stored in the cache, never null.
     * @return data in the read/write format
     */
    Object fromStorage(Object content);

    /**
     * Returns the {@link MediaType} produced by this encoder or null if the storage format is not
     known.
     */
    MediaType getStorageFormat();
}
```

#### 4.7.2. デフォルトのエンコーダー

Data Grid は、キャッシュ設定に応じてエンコーダーを自動的に選択します。以下の表は、複数の設定に使用される内部エンコーダーを示しています。

モード	設定	エンコーダー	説明
組み込み/サーバー	デフォルト	IdentityEncoder	パススルーエンコーダ、変換なし
組み込み	StorageType.OFF_HEAP	GlobalMarshallerEncoder	Data Grid の内部マーシャラーを使用して byte[] に変換します。キャッシュマネージャーで設定されたマーシャラーに委譲できます。

モード	設定	エンコーダー	説明
組み込み	StorageType.BINARY	BinaryEncoder	Data Grid の内部マーシャラーを使用して、プリミティブおよび String を除いて byte[] に変換します。
サーバー	StorageType.OFF_HEAP	IdentityEncoder	リモートクライアントが受信したように byte[] を直接保存

### 4.7.3. プログラムでの上書き

`AdvancedCache` から `.withEncoding()` メソッドバリエーションを呼び出すことで、キーと値の両方に使用されるエンコーディングをプログラムで上書きできます。

たとえば、OFF\_HEAP として設定された以下のキャッシュについて考えてみましょう。

```
// Read and write POJO, storage will be byte[] since for
// OFF_HEAP the GlobalMarshallerEncoder is used internally:
cache.put(1, new Pojo())
Pojo value = cache.get(1)

// Get the content in its stored format by overriding
// the internal encoder with a no-op encoder (IdentityEncoder)
Cache<?,?> rawContent = cache.getAdvancedCache().withEncoding(IdentityEncoder.class);
byte[] marshalled = (byte[]) rawContent.get(1);
```

オーバーライドは、エントリー数のカウントや OFF\_HEAP キャッシュの byte [] のサイズの計算など、キャッシュ内の操作にデコードが不要な場合に役立ちます。

### 4.7.4. カスタムエンコーダーの定義

カスタムエンコーダーは `EncoderRegistry` に登録できます。

#### 注意

キャッシュを起動する前に、クラスターの各ノードで登録が行われていることを確認してください。

gzip で圧縮/解凍するために使用されるカスタムエンコーダを考えてみましょう。

```
public class GzipEncoder implements Encoder {

    @Override
    public Object toStorage(Object content) {
        assert content instanceof String;
        return compress(content.toString());
    }

    @Override
```



```

public Object fromStorage(Object content) {
    assert content instanceof byte[];
    return decompress((byte[]) content);
}

private byte[] compress(String str) {
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GZIPOutputStream gis = new GZIPOutputStream(baos)) {
        gis.write(str.getBytes("UTF-8"));
        gis.close();
        return baos.toByteArray();
    } catch (IOException e) {
        throw new RuntimeException("Unable to compress", e);
    }
}

private String decompress(byte[] compressed) {
    try (GZIPInputStream gis = new GZIPInputStream(new ByteArrayInputStream(compressed));
        BufferedReader bf = new BufferedReader(new InputStreamReader(gis, "UTF-8"))) {
        StringBuilder result = new StringBuilder();
        String line;
        while ((line = bf.readLine()) != null) {
            result.append(line);
        }
        return result.toString();
    } catch (IOException e) {
        throw new RuntimeException("Unable to decompress", e);
    }
}

@Override
public MediaType getStorageFormat() {
    return MediaType.parse("application/gzip");
}

@Override
public boolean isStorageFormatFilterable() {
    return false;
}

@Override
public short id() {
    return 10000;
}
}

```

以下で登録できます。

```

GlobalComponentRegistry registry = cacheManager.getGlobalComponentRegistry();
EncoderRegistry encoderRegistry = registry.getComponent(EncoderRegistry.class);
encoderRegistry.registerEncoder(new GzipEncoder());

```

次に、キャッシュからのデータの読み書きに使用します。

```

AdvancedCache<String, String> cache = ...

```

```
// Decorate cache with the newly registered encoder, without encoding keys (IdentityEncoder)
// but compressing values
AdvancedCache<String, String> compressingCache = (AdvancedCache<String, String>)
cache.withEncoding(IdentityEncoder.class, GzipEncoder.class);

// All values will be stored compressed...
compressingCache.put("297931749", "0412c789a37f5086f743255cfa693dd5");

// ... but API calls deals with String
String stringValue = compressingCache.get("297931749");

// Bypassing the value encoder to obtain the value as it is stored
Object value = compressingCache.withEncoding(IdentityEncoder.class).get("297931749");

// value is a byte[] which is the compressed value
```

## 4.8. トランスコーダーとデータ変換

Data Grid は **org.infinispan.commons.dataconversion.Transcoder** を使用して MediaType 形式間でデータを変換します。

```
public interface Transcoder {

    /**
     * Transcodes content between two different {@link MediaType}.
     *
     * @param content      Content to transcode.
     * @param contentType The {@link MediaType} of the content.
     * @param destinationType The target {@link MediaType} to convert.
     * @return the transcoded content.
     */
    Object transcode(Object content, MediaType contentType, MediaType destinationType);

    /**
     * @return all the {@link MediaType} handled by this Transcoder.
     */
    Set<MediaType> getSupportedMediaTypes();
}
```

### 4.8.1. Converting Data on Demand

タスク、リスナー、マージポリシーなどのカスタムコードを Data Grid にデプロイして実行できます。Data Grid のカスタムコードはデータに直接アクセスできますが、異なるエンドポイントを介して同じデータにアクセスするクライアントとも相互運用する必要があります。たとえば、Hot Rod クライアントがバイナリー形式でデータを読み書きしているときに、カスタムオブジェクトを処理するタスクを作成できます。

この場合、**application/x-protostream** をキャッシュエンコーディングとして設定して、データをバイナリー形式で格納し、異なる MediaType を使用してキャッシュ操作を実行するようにカスタムコードを設定できます。

以下に例を示します。

```

DefaultCacheManager cacheManager = new DefaultCacheManager();

// The cache will store POJO for keys and values
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-java-object");
cfg.encoding().value().mediaType("application/x-java-object");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John", "Doe"));

// Wraps cache using 'application/x-java-object' for keys but JSON for values
Cache<Integer, byte[]> jsonValuesCache = (Cache<Integer, byte[]>)
cache.getAdvancedCache().withMediaType("application/x-java-object", "application/json");

byte[] json = jsonValuesCache.get(1);

```

JSON 形式で値を返します。

```

{
  "_type":"org.infinispan.sample.Person",
  "name":"John",
  "surname":"Doe"
}

```

#### 4.8.2. 組み込み型デロイメントへのトランスコーダーのインストール

Data Grid Server には、デフォルトでトランスコーダーが含まれています。ただし、Data Grid をライブラリーとして実行する場合は、以下をプロジェクトに追加する必要があります。

```
org.infinispan:infinispan-server-core
```

#### 4.8.3. トランスコードおよびエンコーダー

通常、キャッシュ操作に関するデータ変換はないか、1つだけです。

- 組み込みモードまたはサーバーモードを使用したキャッシュでは、デフォルトでは変換されません。
- MediaType は設定されず、OFF\_HEAP または BINARY を使用する組み込みキャッシュのエンコーダベースの変換。
- 複数の REST クライアントと Hot Rod クライアントが異なる形式でデータを送受信するサーバーモードで使用されるキャッシュのトランスコーダベースの変換。これらのキャッシュには、ストレージを記述する MediaType が設定されています。

ただし、高度なユースケースでは、エンコーダーとトランスコーダーの両方を同時に使用できます。

たとえば、マーシャリングされるオブジェクト (jboss marshaller を使用して) コンテンツを格納するキャッシュについて考えてみましょう。セキュリティ上の理由から、プレーンデータが外部ストアに格納されないように、透過的な暗号化レイヤーを追加する必要があります。クライアントは、複数の形式でデータを読み書きできる必要があります。

これは、エンコーディング層に関係なく、ストレージを記述する `MediaType` を使用してキャッシュを設定することで実現できます。

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
```

透過的な暗号化は、たとえば次のように、保存/取得で暗号化/復号化する特別な **エンコーダー** でキャッシュをデコレートすることで追加できます。

```
class Scrambler implements Encoder {

    public Object toStorage(Object content) {
        // Encrypt data
    }

    public Object fromStorage(Object content) {
        // Decrypt data
    }

    @Override
    public boolean isStorageFormatFilterable() {

    }

    public MediaType getStorageFormat() {
        return new MediaType("application", "scrambled");
    }

    @Override
    public short id() {
        //return id
    }
}
```

キャッシュに書き込まれるすべてのデータが暗号化され保存されるようにするには、上記のエンコーダーでキャッシュをデコレートし、このデコレートされたキャッシュですべてのキャッシュ操作を実行する必要があります。

```
Cache<?,?> secureStorageCache =
cache.getAdvancedCache().withEncoding(Scrambler.class).put(k,v);
```

キャッシュを目的の `MediaType` でデコレートすることにより、複数の形式でデータを読み取る機能を追加できます。

```
// Obtain a stream of values in XML format from the secure cache
secureStorageCache.getAdvancedCache().withMediaType("application/xml","application/xml").values
().stream();
```

内部的には、Data Grid は最初にエンコーダー **fromStorage** 操作を適用して、`application/x-jboss-marshalling` 形式のエントリを取得し、次に適切なトランスコーダーを使用して `application/xml` に順次変換を適用します。

## 第5章 JAVA オブジェクトをマーシャルするための DATA GRID の設定

マーシャリングは、Java オブジェクトをバイナリー形式に変換して、ネットワーク経由で転送したり、ディスクに保存したりできるようにします。逆プロセスのアンマーシャリングは、データをバイナリー形式から Java オブジェクトに変換します。

Data Grid は、マーシャリングとアンマーシャリングを実行し、以下を行います。

- クラスターの他の Data Grid ノードにデータを送信します。
- データを永続キャッシュストアに保存します。
- データをバイナリー形式で格納し、デシリアライズ機能を提供します。

### 5.1. サポートされるタイプ

Data Grid は ProtoStream API を使用して、言語に依存しない下位互換性のある形式で、Java オブジェクトをプロトコルバッファ (Protobuf) にエンコードおよびデコードします。

ProtoStream は、キーと値の以下のタイプや、プリミティブタイプの場合にボックス化されていない同等の型を処理できます。

- **byte[]**
- **Byte**
- **文字列**
- **整数**
- **ロング**
- **Double**
- **Float**
- **Boolean**
- **Short**
- **Character**
- **java.util.Date**
- **java.time.Instant**

#### 参照

- [Data Grid ProtoStream library](#)
- [プロトコルバッファ](#)

### 5.2. PROTOSTREAM を使用したユーザータイプのマーシャリング

ユーザータイプは、Data Grid がそのままではサポートしない Java オブジェクトです。ユーザータイプをマーシャリングするには、**SerializationContextInitializer** インターフェイスを実装して Java オブジェクトを記述し、ProtoStream ライブラリーが Protobuf 形式にエンコードし、Data Grid がそれらを送信および保存できるようにします。

### 5.2.1. シリアル化コンテキストイニシャライザーの生成

ProtoStream **SerializationContext** には、Protobuf スキーマからロードされたカスタム Java オブジェクトの Protobuf タイプ定義と、それらのオブジェクトに付随するマーシャラーが含まれています。

Data Grid は、コンパイル時にクラスの Java アノテーションを処理する **protostream-processor** アーティファクトを提供します。プロセッサは、Protobuf スキーマ、マーシャラー、および ProtoStream **SerializationContext** の初期化に使用できる **SerializationContextInitializer** インターフェイスの具体的な実装を生成します。



#### 注記

デフォルトでは、実装名は Impl 接尾辞が付いたアノテーション付きクラス名です。

#### 手順

1. **protostream-processor** 依存関係を **pom.xml** に追加します。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.infinispan.protostream</groupId>
    <artifactId>protostream-processor</artifactId>
    <!--
      This dependency should be declared in the "provided" scope or made "optional"
      because it is a compile-only dependency and is not required at runtime.
      Transitive propagation of this dependency should be also be avoided.
    -->
    <scope>provided</scope>
  </dependency>
</dependencies>
```

2. マーシャリングする Java オブジェクトに **@ProtoField** と **@ProtoFactory** アノテーションを付けます。

#### Author.java

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;
```

```

public class Author {
    @ProtoField(number = 1)
    final String name;

    @ProtoField(number = 2)
    final String surname;

    @ProtoFactory
    Author(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
    // public Getter methods omitted for brevity
}

```

### Book.java

```

import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;
...

public class Book {
    @ProtoField(number = 1)
    final String title;

    @ProtoField(number = 2)
    final String description;

    @ProtoField(number = 3, defaultValue = "0")
    final int publicationYear;

    @ProtoField(number = 4, collectionImplementation = ArrayList.class)
    final List<Author> authors;

    @ProtoFactory
    Book(String title, String description, int publicationYear, List<Author> authors) {
        this.title = title;
        this.description = description;
        this.publicationYear = publicationYear;
        this.authors = authors;
    }
    // public Getter methods omitted for brevity
}

```

3. **SerializationContextInitializer** を拡張し、**@AutoProtoSchemaBuilder** アノテーションが付けられたインターフェイスを定義します。

```

@AutoProtoSchemaBuilder(
    includeClasses = {
        Book.class,
        Author.class,
    },
    schemaFileName = "library.proto", ①
    schemaFilePath = "proto/", ②
)

```

```

    schemaPackageName = "book_sample")
interface LibraryInitializer extends SerializationContextInitializer {
}

```

- 1 生成された **.proto** スキーマファイルに名前を付けます。
- 2 スキーマファイルが生成される **target/classes** の下にパスを設定します。

## 次のステップ

**SerializationContextInitializer** 実装を Data Grid 設定に追加して登録します。

[シリアル化コンテキストイニシャライザーの登録](#) を参照してください。

## 5.2.2. シリアライゼーションコンテキストイニシャライザーの手動実装

場合によっては、Protobuf スキーマを手動で定義し、ProtoStream マーシャラーを実装する必要があります。たとえば、Java オブジェクトクラスを変更してアノテーションを追加できない場合などです。

### 手順

1. マーシャリングする Java オブジェクトの構造化表現を提供する Protobuf スキーマの **.proto** ファイルを作成します。

```

package book_sample;

message Book {
    optional string title = 1;
    optional string description = 2;
    optional int32 publicationYear = 3; // no native Date type available in Protobuf

    repeated Author authors = 4;
}

message Author {
    optional string name = 1;
    optional string surname = 2;
}

```

前述の **.library.proto** ファイルは、**book\_sample** パッケージに含まれる **Book** という名前のエンティティ (Protobuf メッセージタイプ) を定義します。**Book** は、プリミティブ型のいくつかのフィールドと、**Author** メッセージタイプである **authors** という名前のアレイ (Protobuf 反復可能フィールド) を宣言します。

- メッセージをネストできますが、結果的に構造は厳密にツリーであり、グラフではありません。
  - 型の継承はできません。
  - コレクションはサポート対象外ですが、フィールドを繰り返してアレイをエミュレートできます。
2. **org.infinispan.protostream.MessageMarshaller** インターフェイスを使用して、クラスのマーシャラーを実装します。





## 注記

**MessageMarshaller** インターフェイスが非推奨になりました。

次のバージョンの Data Grid は、外部のサードパーティー Java オブジェクトクラスに **@ProtoAdaptor** アノテーションを使用するアダプタークラスを作成できる代替実装を提供します。

### BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class BookMarshaller implements MessageMarshaller<Book> {

    @Override
    public String getTypeName() {
        return "book_sample.Book";
    }

    @Override
    public Class<? extends Book> getJavaClass() {
        return Book.class;
    }

    @Override
    public void writeTo(MessageMarshaller.ProtoStreamWriter writer, Book book) throws
    IOException {
        writer.writeString("title", book.getTitle());
        writer.writeString("description", book.getDescription());
        writer.writeInt("publicationYear", book.getPublicationYear());
        writer.writeCollection("authors", book.getAuthors(), Author.class);
    }

    @Override
    public Book readFrom(MessageMarshaller.ProtoStreamReader reader) throws IOException
    {
        String title = reader.readString("title");
        String description = reader.readString("description");
        int publicationYear = reader.readInt("publicationYear");
        List<Author> authors = reader.readCollection("authors", new ArrayList<>(), Author.class);
        return new Book(title, description, publicationYear, authors);
    }
}
```

### AuthorMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class AuthorMarshaller implements MessageMarshaller<Author> {

    @Override
    public String getTypeName() {
        return "book_sample.Author";
    }
}
```

```

@Override
public Class<? extends Author> getJavaClass() {
    return Author.class;
}

@Override
public void writeTo(MessageMarshaller.ProtoStreamWriter writer, Author author) throws
IOException {
    writer.writeString("name", author.getName());
    writer.writeString("surname", author.getSurname());
}

@Override
public Author readFrom(MessageMarshaller.ProtoStreamReader reader) throws
IOException {
    String name = reader.readString("name");
    String surname = reader.readString("surname");
    return new Author(name, surname);
}
}

```

3. **.proto** スキーマと ProtoStream マーシャラー実装を **SerializationContext** に登録する **SerializationContextInitializer** 実装を作成します。

#### ManualSerializationContextInitializer.java

```

import org.infinispan.protostream.FileDescriptorSource;
import org.infinispan.protostream.SerializationContext;
import org.infinispan.protostream.SerializationContextInitializer;
...

public class ManualSerializationContextInitializer implements SerializationContextInitializer {
    @Override
    public String getProtoFileName() {
        return "library.proto";
    }

    @Override
    public String getProtoFile() throws UncheckedIOException {
        // Assumes that the file is located in a Jar's resources, we must provide the path to the
        library.proto file
        return FileDescriptorSource.getResourceAsString(getClass(), "/" + getProtoFileName());
    }

    @Override
    public void registerSchema(SerializationContext serCtx) {
        serCtx.registerProtoFiles(FileDescriptorSource.fromString(getProtoFileName(),
getProtoFile()));
    }

    @Override
    public void registerMarshallers(SerializationContext serCtx) {
        serCtx.registerMarshaller(new AuthorMarshaller());
        serCtx.registerMarshaller(new BookMarshaller());
    }
}

```

## 次のステップ

**SerializationContextInitializer** 実装を Data Grid 設定に追加して登録します。

[シリアル化コンテキストイニシャライザーの登録](#) を参照してください。

### 5.2.3. シリアル化コンテキストイニシャライザーの登録

Data Grid 設定で **SerializationContextInitializer** 実装を宣言して登録します。

#### 手順

- 以下の例のように、プログラムまたは宣言的に **SerializationContextInitializer** 実装を手動で登録します。

#### プログラムによる設定

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
    .addContextInitializers(new LibraryInitializerImpl(), new SCImpl());
```

#### 宣言型設定

```
<serialization>
  <context-initializer class="org.infinispan.example.LibraryInitializerImpl"/>
  <context-initializer class="org.infinispan.example.another.SCImpl"/>
</serialization>
```

## 5.3. 代替マーシャラー実装の設定

Data Grid は、ProtoStream の代わりに使用できる Marshaller 実装を提供します。また、カスタムマーシャラー実装を使用するように Data Grid を設定することもできます。

### 5.3.1. JBoss Marshalling の使用

JBoss Marshalling はシリアル化ベースのマーシャリングライブラリーであり、以前の Data Grid バージョンではデフォルトのマーシャラーでした。



#### 注記

- Data Grid では、シリアル化ベースのマーシャリングを使用しないでください。代わりに、後方互換性を保証する高性能のバイナリーワイヤー形式である Protostream を使用する必要があります。
- JBoss Marshalling および **AdvancedExternalizer** インターフェイスは非推奨となり、今後のリリースで削除される予定です。しかし、Data Grid は、JBoss Marshalling を使用せずにデータを永続化したときに **AdvancedExternalizer** 実装を無視します。

#### 手順

1. **infinispan-jboss-marshalling** 依存関係をクラスパスに追加します。

2. **GenericJBossMarshaller** を使用するように Data Grid を設定します。
3. Java クラスをデシリアライズ許可リストに追加します。

- プログラムで行う:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
    .marshaller(new GenericJBossMarshaller())
    .whiteList()
    .addRegexps("org.infinispan.example.", "org.infinispan.concrete.SomeClass");
```

- 宣言的に行う:

```
<serialization
marshaller="org.infinispan.jboss.marshalling.common.GenericJBossMarshaller">
<white-list>
<class>org.infinispan.concrete.SomeClass</class>
<regex>org.infinispan.example.*</regex>
</white-list>
</serialization>
```

## 参照

- [Adding Java Classes to Deserialization White Lists](#)
- [AdvancedExternalizer](#)

### 5.3.2. Java シリアライゼーションの使用

Data Grid で Java シリアライゼーションを使用すると、Java オブジェクトが Java **Serializable** インターフェイスを実装する場合にのみ、オブジェクトをマーシャリングできます。

## 手順

1. **JavaSerializationMarshaller** をマーシャラーとして使用するように Data Grid を設定します。
2. Java クラスをデシリアライズ許可リストに追加します。

- プログラムで行う:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
    .marshaller(new JavaSerializationMarshaller())
    .whiteList()
    .addRegexps("org.infinispan.example.", "org.infinispan.concrete.SomeClass");
```

- 宣言的に行う:

```
<serialization
marshaller="org.infinispan.common.marshall.JavaSerializationMarshaller">
<white-list>
<class>org.infinispan.concrete.SomeClass</class>
```

```

<regex>org.infinispan.example.*</regex>
</white-list>
</serialization>

```

## 参照

- [Adding Java Classes to Deserialization White Lists](#)
- [Serializable](#)
- [org.infinispan.commons.marshall.JavaSerializationMarshaller](#)

### 5.3.3. Kryo Marshaller の使用

Data Grid は、Kryo ライブラリーを使用するマーシャリング実装を提供します。

#### Data Grid サーバーの前提条件

Data Grid サーバーで Kryo マーシャリングを使用するには、以下のように Kryo マーシャリング実装のランタイムクラスファイルが含まれる JAR を追加します。

1. Data Grid Maven リポジトリから **infinispan-marshaller-kryo-bundle.jar** をコピーします。
2. JAR ファイルを Data Grid サーバーのインストールディレクトリーにある **server/lib** ディレクトリーに追加します。

#### Data Grid ライブラリーモードの前提条件

Data Grid で Kryo マーシャリングをアプリケーションの埋め込みライブラリーとして使用するには、以下の手順を実行します。

1. **infinispan-marshaller-kryo** 依存関係を **pom.xml** に追加します。

```

<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-marshaller-kryo</artifactId>
  <version>${version.infinispan}</version>
</dependency>

```

2. **org.infinispan.marshaller.kryo.KryoMarshaller** クラスをマーシャラーとして指定します。

```

GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
    .marshaller(new org.infinispan.marshaller.kryo.KryoMarshaller());

```

## 手順

1. **SerializerRegistryService.java** インターフェイスのサービスプロバイダーを実装します。
2. **register(Kryo)** メソッドにすべてのシリアライザー登録を配置します。シリアライザーは、Kryo API を使用して提供された **Kryo** オブジェクトに登録されます。以下に例を示します。

```

kryo.register(ExampleObject.class, new ExampleObjectSerializer())

```

- 以下の範囲内で、デプロイメント JAR ファイルにクラスを実装するためのフルパスを指定します。

```
META-INF/services/org/infinispan/marshaller/kryo/SerializerRegistryService
```

## 参照

- [Adding Java Classes to Deserialization White Lists](#)
- [Kryo on GitHub](#)

### 5.3.4. Protostuff Marshaller の使用

Data Grid は、Protostuff ライブラリーを使用するマーシャリング実装を提供します。

#### Data Grid サーバーの前提条件

Data Grid サーバーで Protostuff マーシャリングを使用するには、以下のように Protostuff マーシャリング実装のランタイムクラスファイルが含まれる JAR を追加します。

- Data Grid Maven リポジトリから **infinispan-marshaller-protostuff-bundle.jar** をコピーします。
- JAR ファイルを Data Grid サーバーのインストールディレクトリーにある **server/lib** ディレクトリーに追加します。

#### Data Grid ライブラリーモードの前提条件

Data Grid で Protostuff マーシャリングをアプリケーションの埋め込みライブラリーとして使用するには、以下の手順を実行します。

- infinispan-marshaller-protostuff** 依存関係を **pom.xml** に追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-marshaller-protostuff</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

- org.infinispan.marshaller.protostuff.ProtostuffMarshaller** クラスをマーシャラーとして指定します。

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
    .marshaller(new org.infinispan.marshaller.protostuff.ProtostuffMarshaller());
```

## 手順

オブジェクトマーシャリングのカスタム Protostuff スキーマを登録するには、以下のいずれかを行います。

- register()** メソッドを呼び出します。

```
RuntimeSchema.register(ExampleObject.class, new ExampleObjectSchema());
```

- **register()** メソッドにすべてのスキーマ登録を配置する **SerializerRegistryService.java** インターフェイスのサービスプロバイダーを実装します。  
その後、以下の範囲内で、デプロイメント JAR ファイルにクラスを実装するためのフルパスを指定します。

```
META-INF/services/org/infinispan.marshaller.protostuff.SchemaRegistryService
```

## 参照

- [Adding Java Classes to Deserialization White Lists](#)
- [Protostuff on GitHub](#)

### 5.3.5. カスタムマーシャラーの使用

Data Grid は、カスタムマーシャラー用に実装できる **Marshaller** インターフェイスを提供します。

## 手順

1. **Marshaller** インターフェイスを実装します。
2. マーシャラーを使用するように Data Grid を設定します。
3. Java クラスをデシリアライズ許可リストに追加します。

- プログラムで行う:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
    .marshaller(new org.infinispan.example.marshall.CustomMarshaller())
    .whiteList().addRegexp("org.infinispan.example.*");
```

- 宣言的に行う:

```
<serialization marshaller="org.infinispan.example.marshall.CustomMarshaller">
  <white-list>
    <class>org.infinispan.concrete.SomeClass</class>
    <regex>org.infinispan.example.*</regex>
  </white-list>
</serialization>
```

## ヒント

カスタムマーシャラーの実装は、起動時に呼び出される **initialize()** メソッドを使用して設定済みの許可リストにアクセスできます。

## 参照

- [Adding Java Classes to Deserialization White Lists](#)
- [org.infinispan.commons.marshall.Marshaller](#)

### 5.3.6. Adding Java Classes to Deserialization White Lists

Data Grid では、セキュリティー上の理由から、任意の Java クラスのデシリアルは許可されません。これは、JSON、XML、およびマーシャリングされた **byte[]** コンテンツが該当します。

システムプロパティーを使用するか、Data Grid 設定で Java クラスを指定して、デシリアライズホワイトリストに Java クラスを追加する必要があります。

## システムプロパティー

```
// Specify a comma-separated list of fully qualified class names
-Dinfinispan.deserialization.whitelist.classes=java.time.Instant,com.myclass.Entity

// Specify a regular expression to match classes
-Dinfinispan.deserialization.whitelist.regexp=.*
```

## 宣言的

```
<cache-container>
  <serialization version="1.0" marshaller="org.infinispan.marshall.TestObjectStreamMarshaller">
    <white-list>
      <class>org.infinispan.test.data.Person</class>
      <regex>org.infinispan.test.data.*</regex>
    </white-list>
  </serialization>
</cache-container>
```



### 注記

デシリアライズホワイトリストに追加する Java クラスは、Data Grid **CacheContainer** に適用され、**CacheContainer** が制御するすべてのキャッシュによってデシリアライズできます。



## 第6章 クラスター化されたロック

クラスター化されたロックは、Data Grid クラスターのノード間で分散され、共有されるデータ構造です。クラスター化されたロックにより、ノード間で同期されるコードを実行できます。

### 6.1. ロック API

Data Grid は、埋め込みモードで Data Grid を使用するとき、クラスター上でコードを同時に実行できる **ClusteredLock** API を提供します。

API は以下で設定されます。

- **ClusteredLock** は、クラスター化されたロックを実装するメソッドを公開します。
- **ClusteredLockManager** は、クラスター化されたロックの定義、設定、取得、および削除を行うメソッドを公開します。
- **EmbeddedClusteredLockManagerFactory** は、**ClusteredLockManager** の実装を初期化します。

#### 所有権

Data Grid は、クラスター内のすべてのノードがロックを使用できるように、**NODE** 所有権をサポートします。

#### 再入可能性

Data Grid のクラスター化ロックは、再入可能ではないため、クラスター内のすべてのノードがロックを取得できますが、ロックを作成したノードのみがロックを解放することができます。

同じ所有者に対して2つの連続したロック呼び出しが送信された場合、最初の呼び出しが使用可能であればロックを取得し、2番目の呼び出しはブロックされます。

#### 参照資料

- [EmbeddedClusteredLockManagerFactory](#)
- [ClusteredLockManager](#)
- [ClusteredLock](#)

### 6.2. クラスター化されたロックの使用

アプリケーションに埋め込まれた Data Grid でクラスター化されたロックを使用する方法について説明します。

#### 前提条件

- **infinispan-clustered-lock** 依存関係を **pom.xml** に追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
</dependency>
```

## 手順

1. キャッシュマネージャーから **ClusteredLockManager** インターフェイスを初期化します。このインターフェイスは、クラスター化されたロックの定義、取得、および削除を行うエントリーポイントです。
2. クラスター化されたロックごとに一意の名前を指定します。
3. **lock.tryLock(1, TimeUnit.SECONDS)** メソッドでロックを取得します。

```
// Set up a clustered Cache Manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();

// Configure the cache mode, in this case it is distributed and synchronous.
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC);

// Initialize a new default Cache Manager.
DefaultCacheManager cm = new DefaultCacheManager(global.build(), builder.build());

// Initialize a Clustered Lock Manager.
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

// Define a clustered lock named 'lock'.
clm1.defineLock("lock");

// Get a lock from each node in the cluster.
ClusteredLock lock = clm1.get("lock");

AtomicInteger counter = new AtomicInteger(0);

// Acquire the lock as follows.
// Each 'lock.tryLock(1, TimeUnit.SECONDS)' method attempts to acquire the lock.
// If the lock is not available, the method waits for the timeout period to elapse. When the lock is
// acquired, other calls to acquire the lock are blocked until the lock is released.
CompletableFuture<Boolean> call1 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
    if (r) {
        System.out.println("lock is acquired by the call 1");
        lock.unlock().whenComplete((nil, ex2) -> {
            System.out.println("lock is released by the call 1");
            counter.incrementAndGet();
        });
    }
});

CompletableFuture<Boolean> call2 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
    if (r) {
        System.out.println("lock is acquired by the call 2");
        lock.unlock().whenComplete((nil, ex2) -> {
            System.out.println("lock is released by the call 2");
            counter.incrementAndGet();
        });
    }
});

CompletableFuture<Boolean> call3 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
```

```

if (r) {
    System.out.println("lock is acquired by the call 3");
    lock.unlock().whenComplete((nil, ex2) -> {
        System.out.println("lock is released by the call 3");
        counter.incrementAndGet();
    });
}
});

CompletableFuture.allOf(call1, call2, call3).whenComplete((r, ex) -> {
    // Print the value of the counter.
    System.out.println("Value of the counter is " + counter.get());

    // Stop the Cache Manager.
    cm.stop();
});

```

### 6.3. ロックの内部キャッシュの設定

クラスター化されたロックマネージャーには、ロック状態を格納する内部キャッシュが含まれます。内部キャッシュは、宣言的またはプログラムのいずれかに設定できます。

#### 手順

1. クラスター化されたロックの状態を保存するクラスター内のノード数を定義します。デフォルト値は **-1** で、値をすべてのノードに複製します。
2. キャッシュの信頼性に以下のいずれかの値を指定します。これは、クラスターがパーティションに分割するか、複数のノードが残った場合にクラスター化ロックがどのように動作するかを制御します。
  - **AVAILABLE**: 任意のパーティションのノードが、ロックで同時に操作することができます。
  - **CONSISTENT**: 大多数のパーティションに属するノードのみが、ロック上で動作できます。これはデフォルト値です。
  - プログラムによる設定

```

import org.infinispan.lock.configuration.ClusteredLockManagerConfiguration;
import org.infinispan.lock.configuration.ClusteredLockManagerConfigurationBuilder;
import org.infinispan.lock.configuration.Reliability;
...

GlobalConfigurationBuilder global =
    GlobalConfigurationBuilder.defaultClusteredBuilder();

final ClusteredLockManagerConfiguration config =
    global.addModule(ClusteredLockManagerConfigurationBuilder.class).numOwner(2).reliability(Reliability.AVAILABLE).create();

DefaultCacheManager cm = new DefaultCacheManager(global.build());

```

```
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);  
clm1.defineLock("lock");
```

- 宣言型設定

```
<?xml version="1.0" encoding="UTF-8"?>  
<infinispan  
  xmlns="urn:infinispan:config:11.0">  
  ...  
  <cache-container default-cache="default">  
    <transport/>  
    <local-cache name="default">  
      <locking concurrency-level="100" acquire-timeout="1000"/>  
    </local-cache>  
  
    <clustered-locks xmlns="urn:infinispan:config:clustered-locks:11.0"  
      num-owners = "3"  
      reliability="AVAILABLE">  
      <clustered-lock name="lock1" />  
      <clustered-lock name="lock2" />  
    </clustered-locks>  
  </cache-container>  
  ...  
</infinispan>
```

#### 参照資料

- [ClusteredLockManagerConfiguration](#)
- [Clustered Locks Configuration Schema](#)

## 第7章 クラスター化カウンター

クラスター化されたカウンターは、Data Grid クラスターのすべてのノードで分散され、共有されるカウンターです。カウンターは異なる整合性レベル (strong および weak) を持つことができます。

strong/weak と一貫性のあるカウンターには個別のインターフェイスがありますが、どちらもその値の更新をサポートし、現在の値を返し、その値が更新されたときにイベントを提供します。このドキュメントでは、ユースケースに最適なものを選択する上で役立つ詳細を以下に示します。

### 7.1. インストールおよび設定

カウンターの使用を開始するには、Maven の **pom.xml** ファイルに依存関係を追加する必要があります。

#### pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
</dependency>
```

このカウンターは、本書で後述する **CounterManager** インターフェイスを介して、Data Grid 設定ファイルまたはオンデマンドを設定できます。**EmbeddedCacheManager** の起動時に、起動時に Data Grid 設定ファイルに設定されたカウンターが作成します。これらのカウンターは Eagerly で開始され、すべてのクラスターのノードで利用できます。

#### configuration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
  <cache-container ...>
    <!-- if needed to persist counter, global state needs to be configured -->
    <global-state>
      ...
    </global-state>
    <!-- your caches configuration goes here -->
    <counters xmlns="urn:infinispan:config:counters:11.0" num-owners="3"
reliability="CONSISTENT">
      <strong-counter name="c1" initial-value="1" storage="PERSISTENT"/>
      <strong-counter name="c2" initial-value="2" storage="VOLATILE">
        <lower-bound value="0"/>
      </strong-counter>
      <strong-counter name="c3" initial-value="3" storage="PERSISTENT">
        <upper-bound value="5"/>
      </strong-counter>
      <strong-counter name="c4" initial-value="4" storage="VOLATILE">
        <lower-bound value="0"/>
        <upper-bound value="10"/>
      </strong-counter>
      <weak-counter name="c5" initial-value="5" storage="PERSISTENT" concurrency-level="1"/>
    </counters>
  </cache-container>
</infinispan>
```

または、プログラムを使用して **GlobalConfigurationBuilder** で以下を行います。

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...;
CounterManagerConfigurationBuilder builder =
globalConfigurationBuilder.addModule(CounterManagerConfigurationBuilder.class);
builder.numOwner(3).reliability(Reliability.CONSISTENT);
builder.addStrongCounter().name("c1").initialValue(1).storage(Storage.PERSISTENT);
builder.addStrongCounter().name("c2").initialValue(2).lowerBound(0).storage(Storage.VOLATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PERSISTENT)
;
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).storage(Storage.VOLATILE);
builder.addWeakCounter().name("c5").initialValue(5).concurrencyLevel(1).storage(Storage.PERSISTENT);
```

一方、このカウンターは、**EmbeddedCacheManager** を初期化した後にいつでも設定することができます。

```
CounterManager manager = ...;
manager.defineCounter("c1",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(1).storage(Storage.PERSISTENT).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(2).lowerBound(0).storage(Storage.VOLATILE).build());
manager.defineCounter("c3",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(3).upperBound(5).storage(Storage.PERSISTENT).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(4).lowerBound(0).upperBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c5",
CounterConfiguration.builder(CounterType.WEAK).initialValue(5).concurrencyLevel(1).storage(Storage.PERSISTENT).build());
```



## 注記

**CounterConfiguration** は変更できず、再利用できます。

カウンターが正常に設定されていると、**defineCounter()** メソッドは **true** を返します。そうでない場合は、**true** を返します。ただし、設定が無効な場合は、メソッドによって **CounterConfigurationException** が発生します。カウンターがすでに定義されているかを調べるには、**isDefined()** メソッドを使用します。

```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
    manager.define("someCounter", ...);
}
```

クラスターごとの属性:

- **num-owners**: クラスター全体で保持するカウンターのコピー数を設定します。数値が小さいほど更新操作は高速になりますが、サポートされるサーバークラッシュの数は少なくなります。正の値である必要があり、デフォルト値は **2** です。

- **reliability**: ネットワークパーティションでカウンターの更新動作を設定します。デフォルト値は **AVAILABLE** で、有効な値は次のとおりです。
  - **AVAILABLE**: すべてのパーティションはカウンター値の読み取りと更新が可能です。
  - **CONSISTENT**: プライマリーパーティション (ノードの大多数) のみがカウンター値の読み取りと更新が可能です。残りのパーティションは、その値の読み取りのみ可能です。

#### カウンターごとの属性:

- **initial-value** [common]: [common]: カウンターの初期値を設定します。デフォルトは **0** (ゼロ) です。
- **storage** [common]: クラスターのシャットダウンおよび再起動時のカウンターの動作を設定します。デフォルト値は **VOLATILE** で、有効な値は次のとおりです。
  - **VOLATILE**: カウンターの値はメモリーでのみ利用できます。この値は、クラスターがシャットダウンすると失われます。
  - **PERSISTENT**: カウンターの値はプライベートおよびローカル永続ストアに保存されます。この値は、クラスターがシャットダウンされたときに保持され、再起動後に復元されません。



#### 注記

オンデマンドおよび **VOLATILE** カウンターは、クラスターのシャットダウン後にその値と設定を失います。再起動後に再度定義する必要があります。

- **lower-bound** [strong]: 強力な一貫性のあるカウンターの下限を設定します。デフォルト値は **Long.MIN\_VALUE** です。
- **upper-bound** [strong]: 強力な一貫性のあるカウンターの上限を設定します。デフォルト値は **Long.MAX\_VALUE** です。



#### 注記

**lower-bound** も **upper-bound** も設定されていない場合は、強力なカウンターは無制限として設定されます。



#### 警告

**initial-value** は、**lower-bound** 以上 **upper-bound** 以下である必要があります。

- **concurrency-level** [weak]: 同時更新の数を設定します。正の値である必要があります、デフォルト値は **16** です。

### 7.1.1. カウンター名のリスト表示

定義されたすべてのカウンターをリスト表示するには、**CounterManager.getCounterNames()** メソッドは、クラスター全体で作成されたすべてのカウンター名のコレクションを返します。

## 7.2. COUNTERMANAGER インターフェイス

**CounterManager** インターフェイスは、カウンターを定義、取得、および削除するエントリーポイントです。

### 埋め込みデプロイメント

**CounterManager** は **EmbeddedCacheManager** の作成を自動的にリッスンし、**EmbeddedCacheManager** ごとのインスタンスの登録を続行します。カウンター状態を保存し、デフォルトのカウンターの設定に必要なキャッシュを開始します。

**CounterManager** の取得は、以下の例のように

**EmbeddedCounterManagerFactory.asCounterManager(EmbeddedCacheManager)** を呼び出すだけです。

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager =
EmbeddedCounterManagerFactory.asCounterManager(manager);
```

### サーバーデプロイメント

Hot Rod クライアントの場合、**CounterManager** は **RemoteCacheManager** に登録されており、以下のように取得できます。

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager = RemoteCounterManagerFactory.asCounterManager(manager);
```

### 7.2.1. CounterManager を介したカウンターの削除

**Strong/WeakCounter** と **CounterManager** でカウンターを削除するのに違いがあります。**CounterManager.remove(String)** は、クラスターからカウンター値を削除し、ローカルカウンターインスタンスのカウンターに登録されているすべてのリスナーを削除します。さらに、カウンターインスタンスは再利用可能ではなくなり、無効な結果が返される可能性があります。

一方で、**Strong/WeakCounter** を削除するとカウンター値のみが削除されます。インスタンスは引き続き再利用でき、リスナーは引き続き動作します。



#### 注記

削除後にアクセスされると、カウンターは再作成されます。

## 7.3. カウンター

カウンターは、strong (**StrongCounter**) または weak (**WeakCounter**) になり、いずれも名前で識別されます。各インターフェイスには特定のインターフェイスがありますが、ロジック (つまり各操作により **CompletableFuture** が返される) を共有しているため、更新イベントが返され、初期値にリセットできます。



非同期 API を使用しない場合は、**sync()** メソッドを介して同期カウンターを返すことができます。API は同じですが、**CompletableFuture** の戻り値はありません。

以下のメソッドは、両方のインターフェイスに共通しています。

```
String getName();
CompletableFuture<Long> getValue();
CompletableFuture<Void> reset();
<T extends CounterListener> Handle<T> addListener(T listener);
CounterConfiguration getConfiguration();
CompletableFuture<Void> remove();
SyncStrongCounter sync(); //SyncWeakCounter for WeakCounter
```

- **getName()** はカウンター名 (identifier) を返します。
- **getValue()** は現在のカウンターの値を返します。
- **reset()** により、カウンターの値を初期値にリセットできます。
- **reset()** はリスナーを登録し、更新イベントを受信します。詳細については、[通知およびイベント](#) セクションをご覧ください。
- **getConfiguration()** はカウンターによって使用される設定を返します。
- **remove()** はクラスターからカウンター値を削除します。インスタンスは引き続き使用でき、リスナーが保持されます。
- **sync()** は同期カウンターを作成します。



#### 注記

削除後にアクセスされると、カウンターは再作成されます。

### 7.3.1. StrongCounter インターフェイス: 一貫性または境界が明確になります。

strong カウンターは、Data Grid キャッシュに保存されている単一のキーを使用して、必要な一貫性を提供します。すべての更新は、その値を更新するためにキーロックの下で実行されます。一方、読み取りはロックを取得し、現在の値を読み取ります。さらに、このスキームではカウンター値をバインドでき、比較および設定/スワップなどのアトミック操作を提供できます。

**StrongCounter** は、**getStrongCounter()** メソッドを使用して **CounterManager** から取得することができます。たとえば、以下のようになります。

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getStrongCounter("my-counter");
```



#### 警告

すべての操作は単一のキーに到達するため、**StrongCounter** は競合レートが高くなります。

**StrongCounter** インターフェイスでは、以下のメソッドを追加します。

```
default CompletableFuture<Long> incrementAndGet() {
    return addAndGet(1L);
}

default CompletableFuture<Long> decrementAndGet() {
    return addAndGet(-1L);
}

CompletableFuture<Long> addAndGet(long delta);

CompletableFuture<Boolean> compareAndSet(long expect, long update);

CompletableFuture<Long> compareAndSwap(long expect, long update);
```

- **incrementAndGet()** はカウンターを1つずつ増分し、新しい値を返します。
- **decrementAndGet()** は、1つずつカウンターをデクリメントし、新しい値を返します。
- **addAndGet()** は、delta をカウンターの値に追加し、新しい値を返します。
- **compareAndSet()** および **compareAndSwap()** は、現在の値が想定される場合にカウンターの値を設定します。



#### 注記

**CompletableFuture** が完了すると、操作は完了とみなされます。



#### 注記

compare-and-set と compare-and-swap の相違点は、操作に成功した場合に、compare-and-set は true を返しますが、compare-and-swap は前の値をか返すことです。戻り値が期待値と同じ場合は、compare-and-swap が正常になります。

### 7.3.1.1. バインドされた StrongCounter

バインドされている場合、上記の更新メソッドはすべて、下限または上限に達すると **CounterOutOfBoundsException** を出力します。例外には、どちら側にバインドが到達したかを確認するための次のメソッドがあります。

```
public boolean isUpperBoundReached();
public boolean isLowerBoundReached();
```

### 7.3.1.2. ユースケース

強力なカウンターは、次の使用例に適しています。

- 各更新後にカウンターの値が必要な場合 (例: クラスター単位の ID ジェネレーターまたはシーケンス)
- バインドされたカウンターが必要な場合は (例: レートリミッター)

### 7.3.1.3. 使用例

```

StrongCounter counter = counterManager.getStrongCounter("unbounded_counter");

// incrementing the counter
System.out.println("new value is " + counter.incrementAndGet().get());

// decrement the counter's value by 100 using the functional API
counter.addAndGet(-100).thenApply(v -> {
    System.out.println("new value is " + v);
    return null;
}).get();

// alternative, you can do some work while the counter is updated
CompletableFuture<Long> f = counter.addAndGet(10);
// ... do some work ...
System.out.println("new value is " + f.get());

// and then, check the current value
System.out.println("current value is " + counter.getValue().get());

// finally, reset to initial value
counter.reset().get();
System.out.println("current value is " + counter.getValue().get());

// or set to a new value if zero
System.out.println("compare and set succeeded? " + counter.compareAndSet(0, 1));

```

以下に、バインドされたカウンターを使用する別の例を示します。

```

StrongCounter counter = counterManager.getStrongCounter("bounded_counter");

// incrementing the counter
try {
    System.out.println("new value is " + counter.addAndGet(100).get());
} catch (ExecutionException e) {
    Throwable cause = e.getCause();
    if (cause instanceof CounterOutOfBoundsException) {
        if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
            System.out.println("ops, upper bound reached.");
        } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
            System.out.println("ops, lower bound reached.");
        }
    }
}

// now using the functional API
counter.addAndGet(-100).handle((v, throwable) -> {
    if (throwable != null) {
        Throwable cause = throwable.getCause();
        if (cause instanceof CounterOutOfBoundsException) {
            if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
                System.out.println("ops, upper bound reached.");
            } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
                System.out.println("ops, lower bound reached.");
            }
        }
    }
}

```

```

    return null;
  }
  System.out.println("new value is " + v);
  return null;
}).get();

```

Compare-and-set と Compare-and-swap の比較例:

```

StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue, newValue;
do {
  oldValue = counter.getValue().get();
  newValue = someLogic(oldValue);
} while (!counter.compareAndSet(oldValue, newValue).get());

```

compare-and-swap では、呼び出しカウンターの呼び出し (**counter.getValue()**) が 1 つ保存されます。

```

StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue = counter.getValue().get();
long currentValue, newValue;
do {
  currentValue = oldValue;
  newValue = someLogic(oldValue);
} while ((oldValue = counter.compareAndSwap(oldValue, newValue).get()) != currentValue);

```

### 7.3.2. WeakCounter インターフェイス: 速度が必要な場合

**WeakCounter** は、カウンターの値を Data Grid キャッシュの複数のキーに保存します作成されたキーの数は **concurrency-level** 属性によって設定されます。各キーはカウンターの値の一部の状態を保存し、同時に更新できます。**StrongCounter** よりも優れた点は、キャッシュの競合率が低いことです。一方、値の読み取りはよりコストが高く、バインドは許可されません。



#### 警告

リセット操作は注意して行う必要があります。これは **アトミック** ではなく、中間値を生成します。これらの値は、読み取り操作および登録されたリスナーによって確認できます。

**WeakCounter** は、**getWeakCounter()** メソッドを使用して **CounterManager** から取得できます。たとえば、以下ようになります。

```

CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getWeakCounter("my-counter");

```

#### 7.3.2.1. weak カウンターインターフェイス

**WeakCounter** は、以下のメソッドを追加します。

```

default CompletableFuture<Void> increment() {
    return add(1L);
}

default CompletableFuture<Void> decrement() {
    return add(-1L);
}

CompletableFuture<Void> add(long delta);

```

これらは `StrongCounter` のメソッドと似ていますが、新しい値は返されません。

### 7.3.2.2. ユースケース

weak カウンターは、更新操作の結果が必要ない場合やカウンターの値があまり必要でないユースケースに最適です。統計の収集は、このようなユースケースの良い例になります。

### 7.3.2.3. 例

以下では、弱いカウンターの使用例を示します。

```

WeakCounter counter = counterManager.getWeakCounter("my_counter");

// increment the counter and check its result
counter.increment().get();
System.out.println("current value is " + counter.getValue());

CompletableFuture<Void> f = counter.add(-100);
//do some work
f.get(); //wait until finished
System.out.println("current value is " + counter.getValue().get());

//using the functional API
counter.reset().whenComplete((aVoid, throwable) -> System.out.println("Reset done " + (throwable
== null ? "successfully" : "unsuccessfully"))).get();
System.out.println("current value is " + counter.getValue().get());

```

## 7.4. 通知およびイベント

strong カウンターと weak カウンターの両方が、更新イベントを受信するためにリスナーをサポートします。リスナーは **CounterListener** を実装する必要があり、これを以下の方法で登録できます。

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

**CounterListener** には以下のインターフェイスがあります。

```

public interface CounterListener {
    void onUpdate(CounterEvent entry);
}

```

返される **Handle** オブジェクトには、**CounterListener** がなくなるときに削除するという主な目的があります。また、処理している **CounterListener** インスタンスにアクセスできます。これには、以下のインターフェイスがあります。

```
public interface Handle<T extends CounterListener> {  
    T getCounterListener();  
    void remove();  
}
```

最後に、**CounterEvent** には、以前と現在の値と状態があります。これには、以下のインターフェイスがあります。

```
public interface CounterEvent {  
    long getOldValue();  
    State getOldState();  
    long getNewValue();  
    State getNewState();  
}
```



### 注記

状態は、非有界である strong カウンターおよび weak カウンターでは常に **State.VALID** になります。**State.LOWER\_BOUND\_REACHED** および **State.UPPER\_BOUND\_REACHED** は有界である strong カウンターのみに有効です。



### 警告

weak カウンター **reset()** 操作は、中間値で複数の通知をトリガーします。

## 第8章 ロックおよび同時実行

Data Grid は、マルチバージョン同時実行制御 (MVCC) を利用します (MVCC)。これは、リレーショナルデータベースや他のデータストアでよく使用される同時実行スキームです。MVCC には、粗粒度の Java 同期や、共有データにアクセスするための JDK ロックに比べて、次のような多くの利点があります。

- 同時リーダーとライターの許可
- リーダーとライターが互いにブロックしない
- 書き込みスキューを検出して処理できる
- 内部ロックのストライピングが可能

### 8.1. 実装の詳細のロック

Data Grid の MVCC 実装では、ロックと同期が最小限に抑えられており、可能な限り `compare-and-swap` などのロックフリー技術やロックフリーのデータ構造などに重点を置いています。これにより、マルチ CPU 環境とマルチコア環境の最適化に役立ちます。

特に、Data Grid の MVCC 実装はリーダーに対して高度に最適化されています。リーダースレッドは、エントリーの明示的なロックを取得せず、代わりに問題のエントリーを直接読み込みます。

一方、ライターは、書き込みロックを取得する必要があります。これにより、エントリーごとに1つの同時書き込みのみが保証されるため、同時ライターはキューイングしてエントリーを変更することになります。

同時読み取りを可能にするため、ライターはエントリーを **MVCCEntry** でラップして、変更する予定のエントリーのコピーを作成します。このコピーは、同時リーダーが部分的に変更された状態を認識できないようにします。書き込みが完了したら、**MVCCEntry.commit()** はデータコンテナーへの変更をフラッシュし、後続のリーダーに変更内容が反映されます。

#### 8.1.1. クラスタ化されたキャッシュでの仕組み

クラスタ化されたキャッシュでは、各キーにキーをロックするノードがあります。このノードはプライマリー所有者と呼ばれます。

##### 8.1.1.1. 非トランザクションキャッシュ

1. 書き込み操作はキーのプライマリー所有者に送信されます。
2. プライマリー所有者はキーをロックしようとします。
  - a. 成功すると、操作が他の所有者に転送されます。
  - b. そうでない場合は、例外が発生します。



#### 注記

操作が条件付きで、プライマリーオーナーで失敗した場合、他のオーナーには転送されません。





## 注記

操作がプライマリ所有者でローカルに実行される場合、最初のステップはスキップされます。

### 8.1.2. トランザクションキャッシュ

トランザクションキャッシュは、楽観的および悲観的ロックモードをサポートします。詳細は、トランザクションのロックを参照してください。

### 8.1.3. 分離レベル

分離レベルは、他のトランザクションと同時に実行されているときに読み取ることができるトランザクションに影響します。詳細は、分離レベルを参照してください。

### 8.1.4. LockManager

**LockManager** は、書き込み用にエントリーをロックするコンポーネントです。**LockManager** は、**LockContainer** を使用して、ロックを検索、保持、作成します。**LockContainers** には、ロックストライピングをサポートするものと、エントリーごとに1つのロックをサポートするものの2つの大きな特徴があります。

### 8.1.5. ロックストライピング

ロックストライピングでは、固定サイズの共有ロックコレクションをキャッシュ全体に使用する必要があります。ロックはエントリーのキーのハッシュコードに基づいてエントリーに割り当てられます。JDKの**ConcurrentHashMap** がロックを割り当てる方法と同様に、これにより、関連性のない可能性のあるエントリーが同じロックによってブロックされる代わりに、拡張性の高い固定オーバーヘッドのロックメカニズムが可能になります。

別の方法は、ロックストライピングを無効にすることです。これは、エントリーごとに **新しい** ロックが作成されることを意味します。このアプローチでは、スループットが高くなる **可能性** がありますが、追加のメモリー使用量やガベージコレクションのチェーンなどのコストがかかります。



## デフォルトのロックストライピング設定

異なるキーのロックが同じロックストライプになってしまうとデッドロックが発生する可能性があるため、ロックストライピングはデフォルトで無効になっています。

ロックストライピングで使用される共有ロックコレクションのサイズは、`<locking />` 設定要素の `concurrencyLevel` 属性を使用して調整できます。

設定例:

```
<locking striping="false|true"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

### 8.1.6. 同時実行レベル

この同時実行レベルは、ストライプロックコンテナのサイズを決定する他に、**DataContainer** の内部



など、関連する JDK **ConcurrentHashMap** ベースのコレクションを調整するためにも使用されます。このパラメーターは、Data Grid でもまったく同じ方法で使用されているため、同時実行レベルの詳細については、JDK **ConcurrentHashMap** Javadocs を参照してください。

設定例:

```
<locking concurrency-level="32"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().concurrencyLevel(32);
```

### 8.1.7. ロックタイムアウト

ロックタイムアウトは、競合するロックを待つ時間 (ミリ秒単位) を指定します。

設定例:

```
<locking acquire-timeout="10000"/>
```

または、以下を実行します。

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

### 8.1.8. 一貫性

(すべての所有者がロックされているのとは対照的に) 単一の所有者がロックされるという事実により、次の一貫性の保証が失われることはありません。キー **K** がノード **{A, B}** に対してハッシュ化され、トランザクション **TX1** が、たとえば、**A** 上の **K** のロックを取得したとします。別のトランザクション **TX2** が **B** (またはその他のノード) 上で開始され、**TX2** が **K** のロックを試みる場合、ロックがすでに **TX1** によって保持されているため、タイムアウトでロックに失敗します。理由は、キー **K** のロックがトランザクションの発生場所に関係なく、常に、確定的に、クラスターの同じノードで取得されるからです。

## 8.2. データのバージョン管理

Data Grid は、simple と external の2つの形式のデータバージョン管理をサポートします。simple バージョン管理は、書き込みスキューチェックのトランザクションキャッシュで使用されます。

external バージョン管理は、Data Grid を Hibernate で使用する場合など、Data Grid 内のデータバージョン管理の外部ソースをカプセル化するために使用され、そのデータバージョン情報をデータベースから直接取得します。

このスキームでは、バージョンに渡すメカニズムが必要になり、オーバーロードされたバージョン **put()** および **putForExternalRead()** が、**AdvancedCache** で提供され、外部データバージョンを取り込みます。その後、これは **InvocationContext** に保管され、コミット時にエントリーに適用されます。



#### 注記

external バージョン管理の場合、書き込みスキューチェックは実行できず、実行されません。

## 第9章 DATA GRID CDI エクステンションの使用

Data Grid は、CDI (Contexts and Dependency Injection) プログラミングモデルと統合し、以下を可能にするエクステンションを提供します。

- CDI Bean および Java EE コンポーネントにキャッシュを設定し、インジェクトします。
- キャッシュマネージャーを設定します。
- キャッシュおよびキャッシュマネージャーレベルのイベントを受信します。
- JCache アノテーションを使用してデータストレージおよび取得を制御します。

### 9.1. CDI 依存関係

以下の依存関係のいずれかで **pom.xml** を更新し、プロジェクトに Data Grid CDI エクステンションを追加します。

#### 埋め込み (Library) モード

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-embedded</artifactId>
</dependency>
```

#### サーバーモード

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-remote</artifactId>
</dependency>
```

### 9.2. 組み込みキャッシュのインジェクト

組み込みキャッシュをインジェクトするために CDI Bean を設定します。

#### 手順

1. キャッシュ修飾子アノテーションを作成します。

```
...
import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache { ❶
}
```

- ❶ **@GreetingCache** 修飾子を作成します。

2. キャッシュ設定を定義するプロデューサーメソッドを追加します。

```

...
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class Config {

    @ConfigureCache("mygreetingcache") ❶
    @GreetingCache ❷
    @Produces
    public Configuration greetingCacheConfiguration() {
        return new ConfigurationBuilder()
            .memory()
            .size(1000)
            .build();
    }
}

```

- ❶ インジェクトするキャッシュに名前を付けます。
- ❷ キャッシュ修飾子を追加します。

3. 必要に応じて、クラスター化されたキャッシュマネージャーを作成するプロデューサーメソッドを追加します。

```

...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

public class Config {

    @GreetingCache ❶
    @Produces
    @ApplicationScoped ❷
    public EmbeddedCacheManager defaultClusteredCacheManager() { ❸
        return new DefaultCacheManager(
            new GlobalConfigurationBuilder().transport().defaultTransport().build());
    }
}

```

- ❶ キャッシュ修飾子を追加します。
- ❷ アプリケーションに対して Bean を 1 度作成します。キャッシュマネージャーを作成するプロデューサーには、複数のキャッシュマネージャーを作成しないように、常に **@ApplicationScoped** アノテーションを含める必要があります。
- ❸ **@GreetingCache** 修飾子にバインドされた新規の **DefaultCacheManager** インスタンスを作成します。



## 注記

キャッシュマネージャーは、ヘビーウェイトオブジェクトです。アプリケーションで複数のキャッシュマネージャーを実行すると、パフォーマンスが低下する可能性があります。複数のキャッシュを挿入する場合は、各キャッシュの修飾子をキャッシュマネージャープロデューサーメソッドに追加するか、修飾子を追加しないでください。

4. **@GreetingCache** 修飾子をキャッシュインジェクションポイントに追加します。

```
...
import javax.inject.Inject;

public class GreetingService {

    @Inject @GreetingCache
    private Cache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

## 9.3. リモートキャッシュの注入

リモートキャッシュを注入するために CDI Bean を設定します。

### 手順

1. キャッシュ修飾子アノテーションを作成します。

```
@Remote("mygreetingcache") ❶
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RemoteGreetingCache { ❷
}
```

- ❶ インジェクトするキャッシュに名前を付けます。
- ❷ **@RemoteGreetingCache** 修飾子を作成します。

2. キャッシュインジェクションポイントに **@RemoteGreetingCache** 修飾子を追加します。

```
public class GreetingService {

    @Inject @RemoteGreetingCache
```

```

private RemoteCache<String, String> cache;

public String greet(String user) {
    String cachedValue = cache.get(user);
    if (cachedValue == null) {
        cachedValue = "Hello " + user;
        cache.put(user, cachedValue);
    }
    return cachedValue;
}
}

```

#### リモートキャッシュをインジェクトするためのヒント

- 修飾子を使用せずにリモートキャッシュをインジェクトできます。

```

...
@Inject
@Remote("greetingCache")
private RemoteCache<String, String> cache;

```

- 複数の Data Grid クラスターがある場合は、クラスターごとに個別のリモートキャッシュマネージャプロデューサーを作成できます。

```

...
import javax.enterprise.context.ApplicationScoped;

public class Config {

    @RemoteGreetingCache
    @Produces
    @ApplicationScoped ❶
    public ConfigurationBuilder builder = new ConfigurationBuilder(); ❷
        builder.addServer().host("localhost").port(11222);
        return new RemoteCacheManager(builder.build());
    }
}

```

- ❶ アプリケーションに対して Bean を 1 度作成します。キャッシュマネージャを作成するプロデューサーには、ヘビーウェイトオブジェクトである複数のキャッシュマネージャが作成されないように、常に **@ApplicationScoped** アノテーションが含まれる必要があります。
- ❷ **@RemoteGreetingCache** 修飾子にバインドされる新しい **RemoteCacheManager** インスタンスを作成します。

## 9.4. JCACHE キャッシングアノテーション

JCache アーティファクトがクラスパスにある場合、以下の JCache キャッシングアノテーションを CDI 管理 Bean で使用できます。

### @CacheResult

メソッド呼び出しの結果をキャッシュします。

## @CachePut

メソッドパラメーターをキャッシュします。

## @CacheRemoveEntry

キャッシュからエントリーを削除します。

## @CacheRemoveAll

キャッシュからすべてのエントリーを削除します。



### 重要

**Target type:**これらの JCache キャッシングアノテーションはメソッドでのみ使用できます。

JCache キャッシュアノテーションを使用するには、アプリケーションの **beans.xml** ファイルでインターセプターを宣言します。

## マネージド環境 (アプリケーションサーバー)

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.InjectedCacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCacheRemoveAllInterceptor</class>
  </interceptors>
</beans>

```

## マネージド外環境 (スタンドアロン)

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
  </interceptors>
</beans>

```

## JCache キャッシングアノテーションの例

以下の例は、**@CacheResult** アノテーションが **GreetingService.greet()** メソッドの結果をキャッシュする方法を示しています。

```
import javax.cache.interceptor.CacheResult;

public class GreetingService {

    @CacheResult
    public String greet(String user) {
        return "Hello" + user;
    }
}
```

JCache アノテーションを使用すると、デフォルトのキャッシュは、アノテーションが付けられたメソッドの完全修飾名をパラメータタイプで使用します。例を以下に示します。

**org.infinispan.example.GreetingService.greet(java.lang.String)**

デフォルト以外のキャッシュを使用するには、以下の例のように、**cacheName** 属性を使用してキャッシュ名を指定します。

```
@CacheResult(cacheName = "greeting-cache")
```

## 9.5. キャッシュおよびキャッシュマネージャイベントの受信

CDI イベントを使用して、キャッシュおよびキャッシュマネージャレベルのイベントを受信します。

- 以下の例のように **@Observes** アノテーションを使用します。

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

    // Cache level events
    private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
        ...
    }

    // Cache manager level events
    private void cacheStarted(@Observes CacheStartedEvent event) {
        ...
    }
}
```

## 第10章 DATA GRID トランザクション

Data Grid は、JTA 準拠のトランザクションを使用し、参加するように設定できます。

または、トランザクションのサポートが無効になっている場合は、JDBC 呼び出しで自動コミットを使用する場合と同等になります。ここでは、すべての変更後に変更がレプリケートされる可能性があります (レプリケーションが有効な場合)。

すべてのキャッシュ操作で Data Grid は以下を行います。

1. スレッドに関連する現在の **トランザクション** を取得します。
2. トランザクションのコミットまたはロールバック時に通知されるように、**XAResource** をトランザクションマネージャーに登録します (登録されていない場合)。

これを実行するには、キャッシュに環境の **TransactionManager** への参照を提供する必要があります。これは通常、**TransactionManagerLookup** インターフェイスの実装のクラス名を使用してキャッシュを設定することで行います。キャッシュが起動すると、このクラスのインスタンスを作成し、**TransactionManager** への参照を返す **getTransactionManager()** メソッドを呼び出します。

Data Grid には複数のトランザクションマネージャールックアップクラスが同梱されます。

### トランザクションマネージャールックアップの実装

- **EmbeddedTransactionManagerLookup**:これは、他の実装が利用できない場合に、埋め込みモードのみに使用する必要がある基本的なトランザクションマネージャーを提供します。この実装は、同時トランザクションおよびリカバリーでは、重大な制限があります。
- **JBossStandaloneJTAManagerLookup**:スタンドアロン環境、または JBoss AS 7 以前、および WildFly 8、9、10 で Data Grid を実行している場合、トランザクションマネージャーのデフォルトとしてこれを選択します。このトランザクションは、**EmbeddedTransactionManager** の不足をすべて解消する **JBoss Transactions** をベースとした本格的なトランザクションマネージャーです。
- **WildflyTransactionManagerLookup**:WildFly 11 以降で Data Grid を実行している場合は、トランザクションマネージャーのデフォルトとしてこれを選択します。
- **GenericTransactionManagerLookup**:これは、最も一般的な Java EE アプリケーションサーバーでトランザクションマネージャーを見つけるルックアップクラスです。トランザクションマネージャーが見つからない場合は、**EmbeddedTransactionManager** がデフォルトの設定になります。

WARN:**DummyTransactionManagerLookup** は 9.0 で非推奨となり、今後削除される予定です。代わりに **EmbeddedTransactionManagerLookup** を使用してください。

初期化すると、**TransactionManager** は **Cache** 自体から取得することもできます。

```
//the cache must have a transactionManagerLookupClass defined
Cache cache = cacheManager.getCache();

//equivalent with calling TransactionManagerLookup.getTransactionManager();
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

### 10.1. トランザクションの設定



トランザクションはキャッシュレベルで設定されます。以下はトランザクションの動作に影響する設定と、各設定属性の簡単な説明になります。

```
<locking
  isolation="READ_COMMITTED"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  reaper-interval="30000"
  recovery-cache="__recoveryInfoCacheName__"
  stop-timeout="30000"
  transaction-manager-
lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
```

プログラムを使用する場合

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
  .isolationLevel(IsolationLevel.READ_COMMITTED);
builder.transaction()
  .lockingMode(LockingMode.OPTIMISTIC)
  .autoCommit(true)
  .completedTxTimeout(60000)
  .transactionMode(TransactionMode.NON_TRANSACTIONAL)
  .useSynchronization(false)
  .notifications(true)
  .reaperWakeUpInterval(30000)
  .cacheStopTimeout(30000)
  .transactionManagerLookup(new GenericTransactionManagerLookup())
  .recovery()
  .enabled(false)
  .recoveryInfoCacheName("__recoveryInfoCacheName__");
```

- **isolation** - 分離レベルを設定します。詳細は、[分離レベル](#) を参照してください。デフォルトは **REPEATABLE\_READ** です。
- **locking** - キャッシュが楽観的または悲観的ロックを使用するかどうかを設定します。詳細は、[トランザクションのロック](#) を参照してください。デフォルトは **OPTIMISTIC** です。
- **auto-commit**: 有効にすると、ユーザーは1回の操作でトランザクションを手動で開始する必要はありません。トランザクションは自動的に起動およびコミットされます。デフォルトは **true** です。
- **complete-timeout** - 完了したトランザクションに関する情報を保持する期間 (ミリ秒単位)。デフォルトは **60000** です。
- **mode**: キャッシュがトランザクションかどうかを設定します。デフォルトは **NONE** です。利用可能なオプションは以下のとおりです。
  - **NONE** - 非トランザクションキャッシュ
  - **FULL\_XA** - リカバリーが有効になっている XA トランザクションキャッシュリカバリーの詳細は、[トランザクションリカバリー](#) を参照してください。

- **NON\_DURABLE\_XA** - リカバリーが無効になっている XA トランザクションキャッシュ。
- **NON\_XA** - XA の代わりに [同期化](#) を介して統合されたトランザクションキャッシュ。詳細は、[同期の登録](#) のセクションを参照してください。
- **BATCH** - バッチを使用して操作をグループ化するトランザクションキャッシュ。詳細は [バッチ処理](#) のセクションを参照してください。
- **notifications** - キャッシュリスナーのトランザクションイベントを有効/無効にします。デフォルトは **true** です。
- **reaper-interval** - トランザクション完了情報をクリーンアップするスレッドが開始する間隔 (ミリ秒単位)。デフォルトは **30000** です。
- **recovery-cache** - リカバリー情報を保存するキャッシュ名を設定します。リカバリーの詳細は、[トランザクションリカバリー](#) を参照してください。デフォルトは **recoveryInfoCacheName** です。
- **stop-timeout** - キャッシュの停止時に進行中のトランザクションを待機する時間 (ミリ秒単位)。デフォルトは **30000** です。
- **transaction-manager-lookup** - **javax.transaction.TransactionManager** への参照を検索するクラスの完全修飾クラス名を設定します。デフォルトは **org.infinispan.transaction.lookup.GenericTransactionManagerLookup** です。

2 フェーズコミット (2PC) が Data Grid に実装される方法、およびロックが取得される方法についての詳細は、以下のセクションを参照してください。設定の詳細については、[設定リファレンス](#) を参照してください。

## 10.2. 分離レベル

Data Grid は、[READ\\_COMMITTED](#) および [REPEATABLE\\_READ](#) の 2 つの分離レベルを提供します。

これらの分離レベルは、リーダーが同時書き込みを確認するタイミングを決定し、**MVCCEntry** の異なるサブクラスを使用して内部的に実装されます。MVCCEntry では、状態がデータコンテナにコミットされる方法が異なります。

以下は、Data Grid のコンテキストの **READ\_COMMITTED** および **REPEATABLE\_READ** の違いを理解する上で役立つ詳細な例です。**READ\_COMMITTED** の場合、同じキーで連続して 2 つの読み取り呼び出しを行うと、キーが別のトランザクションによって更新され、2 つ目の読み取りによって新しい更新値が返されることがあります。

```
Thread1: tx1.begin()
Thread1: cache.get(k) // returns v
Thread2:                               tx2.begin()
Thread2:                               cache.get(k) // returns v
Thread2:                               cache.put(k, v2)
Thread2:                               tx2.commit()
Thread1: cache.get(k) // returns v2!
Thread1: tx1.commit()
```

**REPEATABLE\_READ** では、最終 `get` は引き続き `v` を返します。そのため、トランザクション内で同じキーを複数回取得する場合は、**REPEATABLE\_READ** を使用する必要があります。

ただし、読み取りロックが **REPEATABLE\_READ** に対しても取得されないため、この現象が発生する可能性があります。

```

cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                               tx2.begin()
Thread2:                               cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                               cache.get("B") // returns 2
Thread2:                               tx2.commit()

```

## 10.3. トランザクションのロック

### 10.3.1. 悲観的なトランザクションキャッシュ

ロック取得の観点では、悲観的トランザクションはキーの書き込み時にキーのロックを取得します。

1. ロック要求がプライマリ所有者に送信されます (明示的なロック要求または操作のいずれか)。
2. プライマリの所有者はロックの取得を試みます。
  - a. 成功した場合は、正の応答が返されます。
  - b. そうでない場合は、負の応答が送信され、トランザクションはロールバックされます。

たとえば、以下ようになります。

```

transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();

```

`cache.put(k1,v1)` が返されると、**k1** はロックされ、クラスター内のどこかで実行中の他のトランザクションは、これに書き込むことができません。**k1** の読み取りは引き続き可能です。トランザクションの完了時に **k1** のロックが解放されます (コミットまたはロールバック)。



#### 注記

条件付き操作の場合、検証はオリジネーターで実行されます。

### 10.3.2. 楽観的トランザクションキャッシュ

楽観的トランザクションロックはトランザクションの準備時に取得され、トランザクションのコミット (またはロールバック) まで保持されます。これは、書き込みでローカルロックを取得し、準備中にクラスターのロックが取得される 5.0 デフォルトロックモデルとは異なります。

1. 準備はすべての所有者に送信されます。
2. プライマリの所有者は、必要なロックの取得を試みます。
  - a. ロックに成功すると、書き込みのスキューチェックが実行されます。

- b. 書き込みスキューチェックが成功した場合 (または無効化された場合) は、正の応答を送信します。
- c. それ以外の場合は、負の応答が送信され、トランザクションはロールバックされます。

たとえば、以下のようになります。

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until committed/rolled back.
```



#### 注記

条件付きコマンドの場合、検証は引き続きオリジネーターで実行されます。

### 10.3.3. 悲観的または楽観的トランザクションのどちらが必要か

ユースケースの観点からは、同時に実行されている複数のトランザクション間で多くの競合がない場合は、楽観的トランザクションを使用する必要があります。これは、読み取り時と、コミット時 (書き込みスキューチェックが有効) の間でデータが変更された場合に、楽観的トランザクションがロールバックするためです。

一方、キーでの競合が多く、トランザクションのロールバックがあまり望ましくない場合は、悲観的トランザクションの方が適している可能性があります。悲観的トランザクションは、その性質上、よりコストがかかります。各書き込み操作ではロックの取得に RPC が関係する可能性があります。

## 10.4. スキューの書き込み

書き込みスキューは、2つのトランザクションが独立して同時に同じキーの読み取りと書き込みを行うときに発生します。書き込みスキューの結果、両方のトランザクションは同じキーに対して更新を正常にコミットしますが、値は異なります。

Data Grid は、書き込みスキューチェックを自動的に実行し、楽観的トランザクションで **REPEATABLE\_READ** 分離レベルのデータの一貫性を確保します。これにより、Data Grid はトランザクションの1つを検出し、ロールバックできます。

**LOCAL** モードで動作する場合、書き込みスキューの確認は Java オブジェクト参照に依存して違いを比較します。これにより、書き込みスキューをチェックするための信頼性の高い技術が提供されます。

### 10.4.1. 悲観的トランザクションでのキーへの書き込みロックの強制

悲観的トランザクションでの書き込みスキューを回避するには、**Flag.FORCE\_WRITE\_LOCK** で読み取り時にキーをロックします。



#### 注記

- トランザクション以外のキャッシュでは、**Flag.FORCE\_WRITE\_LOCK** は動作しません。**get()** 呼び出しは、キーの値を読み取りますが、ロックをリモートで取得しません。
- **Flag.FORCE\_WRITE\_LOCK** は、同じトランザクションでエンティティーが後で更新されるトランザクションと併用する必要があります。

`Flag.FORCE_WRITE_LOCK` の例については、以下のコードスニペットを比較してください。

```
// begin the transaction
if (!cache.getAdvancedCache().lock(key)) {
    // abort the transaction because the key was not locked
} else {
    cache.get(key);
    cache.put(key, value);
    // commit the transaction
}
```

```
// begin the transaction
try {
    // throws an exception if the key is not locked.
    cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(key);
    cache.put(key, value);
} catch (CacheException e) {
    // mark the transaction rollback-only
}
// commit or rollback the transaction
```

## 10.5. 例外への対処

`CacheException` (またはそのサブクラス) が JTA トランザクションの範囲内のキャッシュメソッドによって出力される場合、トランザクションは自動的にロールバックに対してマークされます。

## 10.6. 同期の登録

デフォルトでは、Data Grid は `XAResource` を介して、分散トランザクションの最初のクラス参加者として登録します。トランザクションの参加者として Data Grid が必要ではなく、ライフサイクル (準備、完了) によってのみ通知される状況があります (例: Data Grid が Hibernate で 2 次レベルキャッシュとして使用される場合など)。

Data Grid は、`同期` を介したトランザクションのエンリストを許可します。これを有効にするには、`NON_XA` トランザクションモードを使用します。

`Synchronization` には、`TransactionManager` が 1PC で 2PC を最適化できるという利点があります。この場合、他の 1 つのリソースのみがそのトランザクションにエンリストされます (`last resource commit optimization`)。つまり、Hibernate 2 次キャッシュ: Data Grid がコミット時よりも `XAResource` として `TransactionManager` に登録する場合、`TransactionManager` は 2 つの `XAResource` (キャッシュとデータベース) を認識し、この最適化を行いません。2 つのリソース間で調整する必要があるため、tx ログをディスクに書き込む必要があります。一方、Data Grid を `Synchronization` として登録すると、`TransactionManager` はディスクへのログの書き込みを省略します (パフォーマンスが向上)。

## 10.7. バッチ処理

バッチ処理は、トランザクションの原子性といくつかの特性を許可しますが、本格的な JTA または XA 機能は許可しません。多くの場合、バッチ処理は本格的なトランザクションよりもはるかに軽量で安価です。

## ヒント

一般的には、トランザクションの参加者のみが Data Grid クラスターである場合に、バッチ処理 API を使用する必要があります。反対に、トランザクションに複数のシステムが必要な場合に、(**TransactionManager** に関連する)JTA トランザクションを使用する必要があります。たとえば、トランザクションの Hello world! を考慮すると、ある銀行口座から別の銀行口座にお金を転送します。両方の口座が Data Grid 内に保存されている場合は、バッチ処理を使用できます。ある口座がデータベースにあり、もう1つの口座が Data Grid の場合は、分散トランザクションが必要になります。



### 注記

バッチ処理を使用するためにトランザクションマネージャーを定義する必要はありません。

### 10.7.1. API

バッチ処理を使用するようにキャッシュを設定したら、**Cache** で **startBatch()** と **endBatch()** を呼び出して使用します。例:

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

### 10.7.2. バッチ処理と JTA

裏では、バッチ機能が JTA トランザクションを開始し、そのスコープ内のすべての呼び出しがそれに関連付けられます。これには、内部 **TransactionManager** 実装が非常に簡単な (例: リカバリーなし) を使用します。バッチ処理では、以下を取得します。

1. 呼び出し中に取得したロックはバッチが完了するまで保持されます。
2. 変更はすべて、バッチ完了プロセスの一部として、クラスター内でバッチ内に複製されます。バッチの各更新のレプリケーションチャット数を減らします。
3. 同期のレプリケーションまたは無効化が使用された場合は、レプリケーション/無効化の失敗により、バッチがロールバックされます。
4. すべてのトランザクション関連の設定は、バッチ処理にも適用されます。

## 10.8. トランザクションリカバリー



リカバリーはXA トランザクションの機能であり、リソースの不測の事態、場合によってはトランザクションマネージャーの障害を対処し、それに応じてそのような状況から回復します。

### 10.8.1. リカバリーを使用するタイミング

外部データベースに保存されたアカウントから Data Grid に保管されたアカウントに転送される分散トランザクションについて考えてみましょう。**TransactionManager.commit()** が呼び出されると、両方のリソースが正常に完了します (第1フェーズ)。コミット (第2) フェーズでは、データベースは、トランザクションマネージャーからコミットリクエストを受け取る前に、Data Grid の変更を問題なく適用します。この時点では、システムが一貫性のない状態です。お金は外部データベースの口座から取得されますが、まだ Data Grid には表示されません (ロックは2フェーズコミットプロトコルの2番目のフェーズでのみリリースされます)。リカバリーはこの状況に対応することで、データベースと Data Grid の両方のデータが一貫した状態で終了します。

### 10.8.2. 仕組み

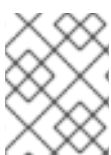
リカバリーはトランザクションマネージャーによって調整されます。トランザクションマネージャーは Data Grid と連携して、手動による介入が必要な未確定のトランザクションのリストを決定し、システム管理者に (電子メール、ログアラートなどを介して) 通知します。このプロセスはトランザクションマネージャーに固有のものですが、通常トランザクションマネージャーで設定が必要になります。

未確定のトランザクション ID を把握すると、システム管理者は Data Grid クラスターに接続し、トランザクションのコミットを再生したり、ロールバックを強制できるようになりました。Data Grid は、この JMX ツールを提供します。これは、[トランザクションのリカバリーおよび調整セクション](#) で広範囲に説明されています。

### 10.8.3. リカバリーの設定

Data Grid では、リカバリーはデフォルトでは**有効になっていません**。無効にすると、**TransactionManager** は Data Grid と動作しないため、インダウト状態のトランザクションを決定できません。[トランザクションの設定](#) セクションでは、その設定を有効にする方法を示しています。

注記: **recovery-cache** 属性は必須ではなく、キャッシュごとに設定されます。



#### 注記

リカバリーが機能するには、完全なXA トランザクションが必要であるため、**mode** を **FULL\_XA** に設定する必要があります。

#### 10.8.3.1. JMX サポートの有効化

リカバリー JMX サポートの管理に JMX を使用できるようにするには、明示的に有効にする必要があります。

### 10.8.4. リカバリーキャッシュ

未確定のトランザクションを追跡し、それらに回答できるようにするために、Data Grid は将来の使用のためにすべてのトランザクション状態をキャッシュします。この状態は、未確定のトランザクションに対してのみ保持され、コミット/ロールバックフェーズが完了した後、正常に完了したトランザクションに対しては削除されます。

この未確定のトランザクションデータはローカルキャッシュ内に保持されます。これにより、データが大きくなりすぎた場合に、キャッシュローダーを介してこの情報をディスクにスワップするように設定できます。このキャッシュは、**recovery-cache** 設定属性を介して指定できます。指定のない場合は、

Data Grid がローカルキャッシュを設定します。

リカバリーが有効になっているすべての Data Grid キャッシュ間で同じリカバリーキャッシュを共有することは可能です (必須ではありません)。デフォルトのリカバリーキャッシュが上書きされた場合、指定のリカバリーキャッシュは、キャッシュ自体が使用するものとは異なるトランザクションマネージャーを返す `TransactionManagerLookup` を使用する必要があります。

### 10.8.5. トランザクションマネージャーとの統合

これはトランザクションマネージャーに固有のものです。通常トランザクションマネージャーは `XAResource.recover()` を呼び出すために `XAResource` 実装への参照が必要になります。Data Grid `XAResource` の以下の API への参照を取得するには、以下を行います。

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

トランザクションを実行するプロセスとは異なるプロセスで復元を実行することが一般的です。

### 10.8.6. 調整

トランザクションマネージャーは、システム管理者に未確定のトランザクションについて独自の方法で通知します。この段階では、システム管理者がトランザクションの XID(バイトアレイ) を把握していることを前提としています。

通常のリカバリーフローは以下のとおりです。

- **ステップ 1:**システム管理者は、JMX を介して Data Grid サーバーに接続し、未確定のトランザクションをリスト表示します。以下のイメージは、未確定のトランザクションを持つ Data Grid ノードに接続する JConsole を示しています。



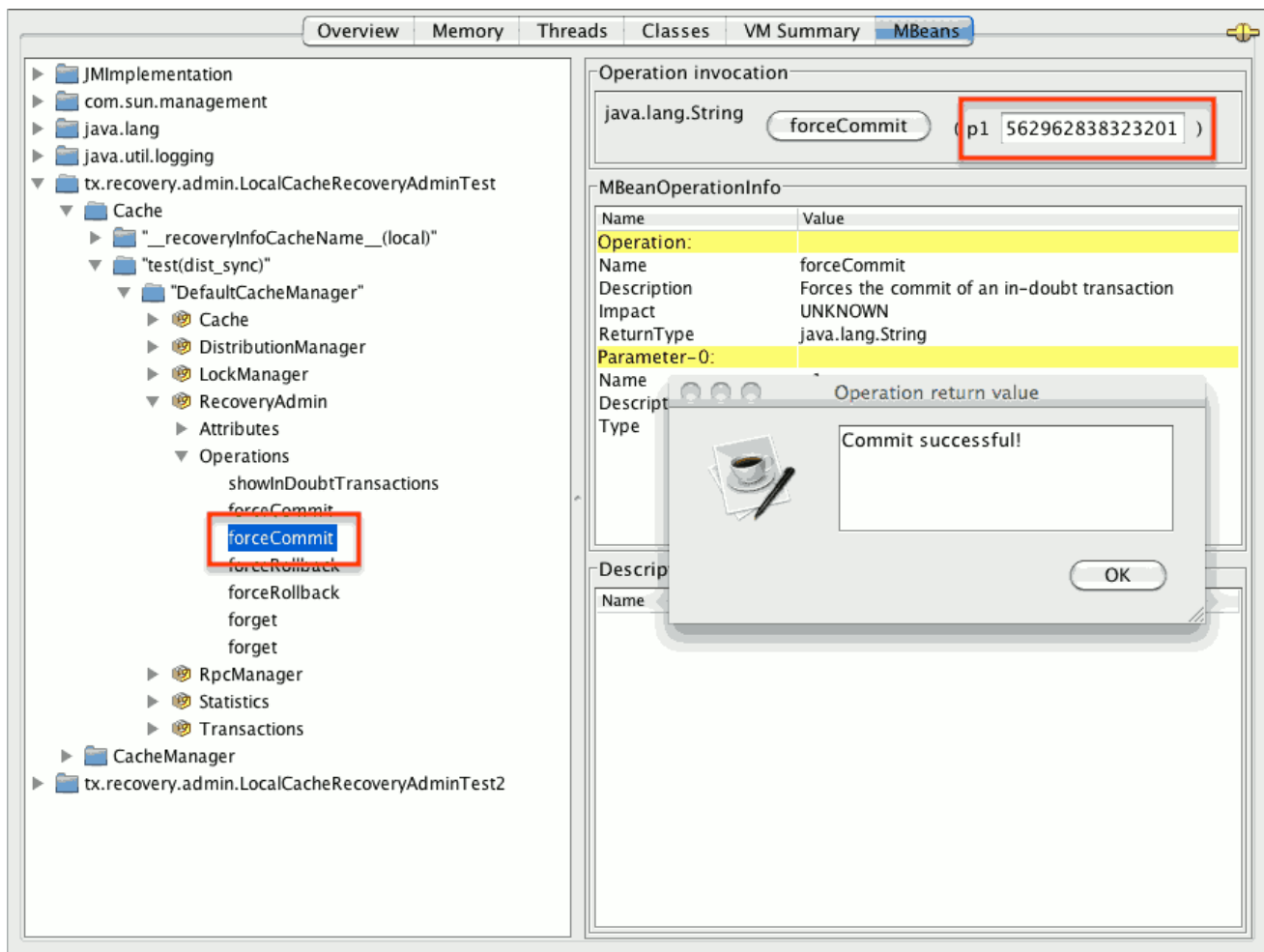
図10.1 未確定のトランザクションの表示

The screenshot shows the JMX console interface. On the left, the MBean tree is expanded to show the `RecoveryAdmin` MBean under the `Cache` component. The `showInDoubtTransactions` operation is selected. On the right, the `Operation invocation` panel shows the operation name `showInDoubtTransactions` and its return value. The return value is a list of transaction objects, with the first one highlighted: `[120-5674-21-1174918-6974-103-3529 >], internalId = 562962838323201, status = [ _PREPARED_ ]`. Annotations with arrows point to the `RecoveryAdmin` MBean, the `showInDoubtTransactions` operation, the transaction ID (labeled `XID`), the `internalId` field, and the `status` field (labeled `Current status of the in-doubt transaction`).

未確定の各トランザクションのステータスが表示されます (この例では **PREPARED** です)。status フィールドに複数の要素が存在する可能性があります。たとえば、トランザクションが特定ノードでコミットされていても、それらのノードでコミットされない場合は **PREPARED** および **COMMITTED** です。

- **ステップ 2:** システム管理者は、トランザクションマネージャーから受け取った XID を数字で表した Data Grid 内部 ID に視覚的にマッピングします。XID(バイトアレイ) は、JMX ツール (JConsole など) に渡して Data Grid 側で再アセンブルされるため、このステップが必要です。
- **ステップ 3:** システム管理者は、内部 ID に基づいて、対応する jmx 操作を介してトランザクションのコミット/ロールバックを強制的に実行します。以下のイメージは、内部 ID に基づいてトランザクションのコミットを強制することで取得します。

図10.2 コミットの強制



## ヒント

上記のすべての JMX 操作は、トランザクションの発信場所に関係なく、任意のノードで実行できます。

### 10.8.6.1. XID に基づくコミット/ロールバックの強制

未確定のトランザクションのコミット/ロールバックの強制を行う XID ベースの JMX 操作も使用できます。これらのメソッドはトランザクションに関連する番号ではなく、XID を記述する byte[] アレイを受け取ります (前述のステップ 2 で説明)。これらは、たとえば、特定の未確定トランザクションの自動完了ジョブを設定する場合に役立ちます。このプロセスはトランザクションマネージャーのリカバリーにプラグインされ、トランザクションマネージャーの XID オブジェクトにアクセスできます。

### 10.8.7. 詳細

[リカバリー設計ドキュメント](#) では、トランザクションリカバリー実装の内部について詳しく説明しています。

## 第11章 インデックス化および検索

Data Grid は、Java POJOs として保存されたキャッシュ値、または [プロトコルバッファ](#) としてエンコードされるオブジェクトとしてインデックス化および検索できる検索 API を提供します。

### 11.1. 概要

検索は [ライブラリーモード](#) と [クライアント/サーバーモード](#) (Java、C#、Node.js、その他のクライアントの場合) の両方で可能であり、Data Grid は [ApacheLucene](#) を使用してデータをインデックス化でき、広い範囲のデータ取得ユースケースをカバーするために効率的な [フルテキスト](#) 対応の検索エンジンを提供します。

インデックス設定は、スキーマ定義に依存し、Data Grid はライブラリーモードではアノテーションが付いた Java クラスを使用でき、リモートクライアントでは protobuf スキーマを使用できます。protobuf で標準化することで、Data Grid は Java クライアントと非 Java クライアントとの間の完全な相互運用性を可能にします。

Data Grid には、文字列ベースで全文検索のサポートを追加する [Ickle](#) と呼ばれる独自のクエリ言語があります。Ickle は、インデックス化されたデータ、部分的にインデックス化されたデータ、またはインデックスなしのデータに対する検索をサポートします。

最後に、Data Grid は [継続的なクエリ](#) をサポートします。これは、他の API とは逆に動作します。クエリの作成、クエリの実行、結果の取得の代わりに、クライアントは、クラスター内のデータの変更に応じて継続的に評価されるクエリを登録でき、変更されたデータがクエリと一致するたびに通知を生成します。

### 11.2. エントリー値のインデックス化

Data Grid のエントリー値をインデックス化すると、検索パフォーマンスが大幅に改善し、フルテキストクエリを実行できるようになります。しかし、インデックス処理は、Data Grid クラスターの書き込みスループットを低下させる可能性があります。このため、キャッシュモードやユースケースに応じて、ストラテジーを使用してクエリのパフォーマンスを最適化する計画を立てる必要があります。 [クエリパフォーマンスガイド](#) についての詳細

#### 11.2.1. 設定

XML によるインデックス化を有効にするには、`<indexing>` 要素をキャッシュ設定に追加し、インデックス化されるエンティティを指定し、オプションで追加のプロパティを渡す必要があります。



#### 注記

`enabled` 属性のデフォルト値が XSD スキーマで `"false"` として定義されている場合でも、`enabled` 属性を省略する `<indexing>` 要素が存在すると、便宜上、インデックスが自動的に有効になります。プログラムによる設定では、`enabled()` を使用する必要があります。

#### 宣言的に

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing>
        <indexed-entities>
```

```

    <indexed-entity>com.acme.Book</indexed-entity>
  </indexed-entities>
  <property name="property.name">some value</property>
</indexing>
</replicated-cache>
</cache-container>
</infinispan>

```

## プログラムで

```

import org.infinispan.configuration.cache.*;

ConfigurationBuilder cacheCfg = ...
cacheCfg.indexing().enable()
    .addIndexedEntity(Book.class)
    .addProperty("property name", "property value")

```

### 11.2.2. インデックス化されたエンティティの指定

インデックス化タイプを宣言することが推奨されます。これは、次の Data Grid バージョンで必須であるためです。

#### 宣言的に

```

<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing>
        <indexed-entities>
          <indexed-entity>com.acme.query.test.Car</indexed-entity>
          <indexed-entity>com.acme.query.test.Truck</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>

```

## プログラムで

```

cacheCfg.indexing()
    .addIndexedEntity(Car.class)
    .addIndexedEntity(Truck.class)

```

キャッシュが protobuf を保存する場合は、インデックス化されたタイプは protobuf スキーマで宣言された **Message** である必要があります。たとえば、以下のスキーマの場合は以下のようになります。

```

package book_sample;

message Book {
  optional string title = 1;
  optional string description = 2;
  optional int32 publicationYear = 3; // no native Date type available in Protobuf

```

```

    repeated Author authors = 4;
  }

  message Author {
    optional string name = 1;
    optional string surname = 2;
  }

```

設定は以下になるはずでず。

```

<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="books">
      <indexing>
        <indexed-entities>
          <indexed-entity>book_sample.Book</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>

```

### 11.2.3. インデックスストレージ

Data Grid は、インデックスをファイルシステムまたはメモリー (**local-heap**) に保存できます。ファイルシステムが推奨され、デフォルト設定であり、メモリーインデックスは、再起動後も保持する必要のない中小規模のインデックスにのみ使用する必要があります。

#### ファイルシステムインデックスの設定:

```

<replicated-cache name="myCache">
  <indexing>
    <indexed-entities>
      <indexed-entity>com.acme.Book</indexed-entity>
    </indexed-entities>
    <!-- Optional: this is the default setting -->
    <property name="default.directory_provider">filesystem</property>
    <!-- Optional: define base folder for indexes -->
    <property name="default.indexBase">${java.io.tmpdir}/baseDir</property>
  </indexing>
</replicated-cache>

```

#### メモリーインデックスの設定:

```

<replicated-cache name="myCache">
  <indexing>
    <indexed-entities>
      <indexed-entity>com.acme.Book</indexed-entity>
    </indexed-entities>
    <property name="default.directory_provider">local-heap</property>
  </indexing>
</replicated-cache>

```

## 11.2.4. インデックスマネージャー

Data Grid は、内部的に Index Manager というコンポーネントを使用して、新規データがインデックスに適用される方法や、データが検索に表示されるタイミングを制御します。

デフォルトの Index Manager **directory-based** は、データがキャッシュに書き込まれるとすぐにインデックスに書き込みます。欠点は、インデックスに対してフラッシュと呼ばれる一定のコストのかかる操作を実行する必要があるため、特に大量の書き込みを行うシナリオでキャッシュの書き込みが大幅に遅くなる可能性があることです。

**near-real-time** インデックスマネージャーはデフォルトのインデックスマネージャーと似ていますが、Lucene の Near-Real-Time 機能を利用します。基になるストアへのインデックスのフラッシュ頻度が少ないため、書き込みパフォーマンスが向上します。欠点は、シャットダウンが適切に行われない場合に、フラッシュされていないインデックスの変更が失われることです。**local-heap** または **filesystem** と併用できます。

**local-heap** を使用した例:

```
<replicated-cache name="default">
  <indexing>
    <property name="default.indexmanager">near-real-time</property>
    <property name="default.directory_provider">local-heap</property>
  </indexing>
</replicated-cache>
```

**filesystem** を使用した例:

```
<replicated-cache name="default">
  <indexing>
    <property name="default.indexmanager">near-real-time</property>
  </indexing>
</replicated-cache>
```

## 11.2.5. インデックスの再構築

インデックスを再構築すると、キャッシュに保存されているデータから再構築されます。インデックス付きタイプやアナライザーの定義などを変更する場合は、インデックスを再構築する必要があります。同様に、何らかの理由でインデックスが削除されている場合は、インデックスを再構築する必要がある場合があります。グリッド内のすべてのデータを再処理する必要があるため、時間がかかる場合があることに注意してください。

```
Indexer indexer = Search.getIndexer(cache);
CompletionStage<Void> future = index.run();
```

## 11.3. 検索

Ickle クエリー言語を使用して、ライブラリーおよびリモートクライアント/サーバーモードの両方でリレーショナルおよびフルテキストクエリーを作成します。

API を使用するには、まず QueryFactory をキャッシュに取得してから、**.create()** メソッドを呼び出し、クエリーで使用する文字列を渡します。各 **QueryFactory** インスタンスは **Search** と同じ **Cache** インスタンスにバインドされますが、それ以外の場合は、複数のクエリーを並行して作成するために使用できるステートレスおよびスレッドセーフオブジェクトになります。

以下に例を示します。

```
// Remote Query, using protobuf
QueryFactory qf = org.infinispan.client.hotrod.Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");

// Embedded Query using Java Objects
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache);
Query q = qf.create("from com.acme.Book where price > 20");

// Execute the query
QueryResult<Book> queryResult = q.execute();
```



### 注記

クエリーは常に単一のエンティティタイプをターゲットにし、単一のキャッシュの内容に対して評価されます。複数のキャッシュでクエリーを実行したり、複数のエンティティタイプ (結合) を対象とするクエリーを作成したりすることは、サポートされていません。

クエリーの実行と結果のフェッチは、**Query**オブジェクトの **run()** メソッドを呼び出すのと同じくらい簡単です。実行後に、同じインスタンスで **run()** を呼び出すと、クエリーを再実行します。

#### 11.3.1. ページネーション

**Query.maxResults(int maxResults)** を使用して、返される結果の数を制限することができます。これを **Query.startOffset(long startOffset)** と組み合わせて使用すると、結果セットのページネーションを実現できます。

```
// sorted by year and match all books that have "clustering" in their title
// and return the third page of 10 results
Query<Book> query = queryFactory.create("FROM com.acme.Book WHERE title like '%clustering%'
ORDER BY year").startOffset(20).maxResults(10)
```

#### 11.3.2. ヒット数

**QueryResult** オブジェクトには、ページネーションパラメーターに関係なく、クエリーの結果の合計数を返すための **.hitCount()** メソッドがあります。ヒット数は、パフォーマンス上の理由から、インデックス付きクエリーでのみ使用できます。

#### 11.3.3. 反復

**Query** オブジェクトには、結果を遅延して取得するための **.iterator()** メソッドがあります。使用後に閉じる必要がある **CloseableIterator** のインスタンスを返します。



### 注記

リモートクエリーの反復サポートは現在制限されています。反復する前に、最初にすべてのエンタリーをクライアントにフェッチするためです。

#### 11.3.4. 名前付きクエリーパラメーターの使用



実行ごとに新しい Query オブジェクトを作成する代わりに、実行前に実際の値に置き換えることができる名前付きパラメーターをクエリーに含めることができます。これにより、クエリーを1度定義し、複数回効率的に実行できます。パラメーターは、Operator の右側でのみ使用でき、通常の定数値ではなく、**org.infinispan.query.dsl.Expression.param(String paramName)** メソッドによって生成されたオブジェクトを Operator に提供することで、クエリーの作成時に定義されます。パラメーターが定義されたら、以下の例に示すように **Query.setParameter(parameterName, value)** または **Query.setParameters(parameterMap)** のいずれかを呼び出すことで設定できます。

```
QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors and publication years
Query<Book> query = queryFactory.create("SELECT title FROM com.acme.Book WHERE author =
:authorName AND publicationYear = :publicationYear").build();

// Set actual parameter values
query.setParameter("authorName", "Doe");
query.setParameter("publicationYear", 2010);

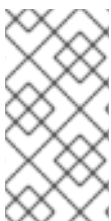
// Execute the query
List<Book> found = query.list();
```

または、実際のパラメーター値のマップを指定して、複数のパラメーターを一度に設定することもできます。

### 複数の名前付きパラメーターを一度に設定する

```
Map<String, Object> parameterMap = new HashMap<>();
parameterMap.put("authorName", "Doe");
parameterMap.put("publicationYear", 2010);

query.setParameters(parameterMap);
```



#### 注記

クエリーの解析、検証、および実行計画の作業の大部分は、パラメーターでのクエリーの最初の実行時に実行されます。この作業は後続の実行時には繰り返し行われなため、クエリーパラメーターではなく定数値を使用した同様のクエリーの場合よりもパフォーマンスが向上します。

### 11.3.5. Ickle クエリー言語パーサー構文

Ickle クエリー言語は、フルテキスト用のいくつかのエクステンションを持つ JPQL クエリー言語の小さなサブセットです。

パーサー構文には、以下のような重要なルールがあります。

- 空白は重要ではありません。
- フィールド名ではワイルドカードはサポートされません。
- デフォルトのフィールドがないため、フィールド名またはパスは必ず指定する必要があります。
- **&&** および **||** は、フルテキストと JPA 述語の両方で、**AND** または **OR** の代わりに使用できます。



- **!**は **NOT** の代わりに使用できます。
- 足りないブール値 Operator は **OR** として解釈されます。
- 文字列の用語は、一重引用符または二重引用符で囲む必要があります。
- ファジー性とブースティングは任意の順序で受け入れられず、常にファジー性が最初になります。
- **<>** の代わりに **!=** が許可されます。
- ブーディングは、**>**、**>=**、**<**、**←** Operator には適用できません。同じ結果を達成するために範囲を使用することができます。

### 11.3.5.1. Operator のフィルタリング

Ickle はインデックス化されたフィールドとインデックス化されていないフィールドの両方に使用できる多くの Operator のフィルタリングをサポートします。

Operator	説明	例
in	左のオペランドが引数として指定された値のコレクションからの要素のいずれかと等しいことを確認します。	FROM Book WHERE isbn IN ('ZZ', 'X1234')
like	(文字列として想定される) 左側の引数が、JPA ルールに準拠するワイルドカードパターンと一致することを確認します。	FROM Book WHERE title LIKE '%Java%'
=	左側の引数が指定の値と完全に一致することを確認します。	FROM Book WHERE name = 'Programming Java'
!=	左側の引数が指定の値とは異なることを確認します。	FROM Book WHERE language != 'English'
>	左側の引数が指定の値よりも大きいことを確認します。	FROM Book WHERE price > 20
>=	左側の引数が指定の値以上であることを確認します。	FROM Book WHERE price >= 20
<	左側の引数が指定の値未満であることを確認します。	FROM Book WHERE year < 2012
←	左側の引数が指定の値以下であることを確認します。	FROM Book WHERE price ← 50
between	左側の引数が指定された範囲の制限の間にあることを確認します。	FROM Book WHERE price BETWEEN 50 AND 100

### 11.3.5.2. ブール値の条件

以下の例では、複数の属性条件を論理結合 (**and**) および非結合 (**or**) 演算子と組み合わせて、より複雑な条件を作成する方法を示しています。ブール値演算子のよく知られる演算子の優先順位ルールがここで適用されるため、Operator の順序は関連性がありません。ここで、**or** が最初に呼び出された場合でも、**and** Operator の優先順位は **or** よりも高くなります。

```
# match all books that have "Data Grid" in their title
# or have an author named "Manik" and their description contains "clustering"

FROM com.acme.Book WHERE title LIKE '%Data Grid%' OR author.name = 'Manik' AND description
like '%clustering%'
```

ブール値の否定は論理演算子の中で優先され、次の単純な属性条件にのみ適用されます。

```
# match all books that do not have "Data Grid" in their title and are authored by "Manik"
FROM com.acme.Book WHERE title != 'Data Grid' AND author.name = 'Manik'
```

### 11.3.5.3. ネストされた条件

論理演算子の優先順位の変更は、括弧を使用して行います。

```
# match all books that have an author named "Manik" and their title contains
# "Data Grid" or their description contains "clustering"
FROM com.acme.Book WHERE author.name = 'Manik' AND ( title like '%Data Grid%' OR description
like '% clustering%')
```

### 11.3.5.4. 属性の選択

一部のユースケースでは、属性のごく一部のみがアプリケーションによって実際に使用されている場合、特にドメインエンティティーにエンティティーが埋め込まれている場合、ドメインオブジェクト全体を返すのはやり過ぎです。クエリー言語を使用すると、プロジェクションを返す属性 (または属性パス) のサブセットを指定できます。デプロイメントが使用される場合、**QueryResult.list()** はドメインエンティティー全体を返しません、**Object[]** の **List** (プロジェクト化された属性に対応する配列) を返します。

```
# match all books that have "Data Grid" in their title or description
# and return only their title and publication year
SELECT title, publicationYear FROM com.acme.Book WHERE title like '%Data Grid%' OR
description like '%Data Grid%'
```

### 11.3.5.5. ソート

1つ以上の属性または属性パスに基づいて結果の順序は **ORDER BY** 句で行われます。複数の並べ替え基準が指定されている場合は、順序によって優先順位が決まります。

```
# match all books that have "Data Grid" in their title or description
# and return them sorted by the publication year and title
FROM com.acme.Book WHERE title like '%Data Grid%' ORDER BY publicationYear DESC, title
ASC
```

### 11.3.5.6. グループ化およびアグリゲーション

Data Grid には、グループ化フィールドのセットに従ってクエリ結果をグループ化し、各グループに分類される値のセットに集計関数を適用することにより、各グループからの結果の集計を構築する機能があります。グループ化と集計は、プロジェクショクエリ (SELECT 句に1つ以上のフィールドがあるクエリ) にのみ適用できます。

サポートされる集約は avg、sum、count、max および min です。

グループ化フィールドセットは **GROUP BY** 句で指定し、グループ化フィールドの定義に使用される順番は関係ありません。プロジェクションで選択されたすべてのフィールドは、グループ化フィールドであるか、以下で説明するグループ化関数の1つを使用して集約される必要があります。Projection フィールドは集約され、同時にグループ化に使用できます。グループ化フィールドのみを選択し、集計フィールドは選択しないクエリは有効です。例:ブックマークは作成者別にグループ化し、それらをカウントします。

```
SELECT author, COUNT(title) FROM com.acme.Book WHERE title LIKE '%engine%' GROUP BY author
```



### 注記

選択したすべてのフィールドに集計関数が適用され、グループ化にフィールドが使用されないプロジェクショクエリが許可されます。この場合、集計は、単一のグローバルグループが存在するようにグローバルに計算されます。

#### 11.3.5.7. 集約

以下の集約関数をフィールドに適用できます。

- **avg()** - 一連の数字の平均を計算します。許可される値は、**java.lang.Number** のプリミティブ番号およびインスタンスです。結果は **java.lang.Double** で表されます。null 以外の値がない場合、結果は代わりに **null** になります。
- **avg()** - null 以外の行の数をカウントし、**java.lang.Long** を返します。null 以外の値がない場合、結果は代わりに **0** になります。
- **max()** - 見つかった最も大きな値を返します。許可される値は **java.lang.Comparable** のインスタンスである必要があります。null 以外の値がない場合、結果は代わりに **null** になります。
- **min()** - 見つかった最小値を返します。許可される値は **java.lang.Comparable** のインスタンスである必要があります。null 以外の値がない場合、結果は代わりに **null** になります。
- **sum()** - 数字のセットの合計を計算します。null 以外の値がない場合、結果は代わりに **null** になります。以下の表は、指定のフィールドに基づいて返されるタイプを示しています。

表11.1 テーブル合計戻り値のタイプ

フィールドタイプ	戻り値のタイプ
Integral (BigInteger 以外)	ロング
Float または Double	Double
BigInteger	BigInteger

フィールドタイプ	戻り値のタイプ
BigDecimal	BigDecimal

### 11.3.5.8. グループ化および集計を使用したクエリーの評価

集計クエリーには、通常のクエリーのようにフィルター条件を含めることができます。フィルタリングは、グループ化操作の前後の2つのステージで実行できます。グループ化操作の実行前に **groupBy()** メソッドを起動する前に定義されたフィルター条件はすべて、キャッシュエントリーに直接 (最終的なデプロイメントではなく) キャッシュエントリーに適用されます。これらのフィルター条件は、照会されたエンティティタイプの任意のフィールドを参照でき、グループ化ステージの入力となるデータセットを制限することを目的としています。**groupBy()** メソッドの呼び出し後に定義されたフィルター条件はすべて、デプロイメントおよびグループ化操作の結果がデプロイメントされます。このフィルター条件は、**groupBy()** フィールドまたは集約されたフィールドのいずれかを参照できます。select 句で指定されていない集約フィールドを参照することは許可されています。ただし、非集計フィールドと非グループ化フィールドを参照することは禁止されています。このフェーズでフィルタリングすると、プロパティーに基づいてグループの数が減ります。通常のクエリーと同様に、並べ替えも指定できます。順序付け操作は、グループ化操作後に実行され、**groupBy()** フィールドまたは集約されたフィールドのいずれかを参照できます。

### 11.3.5.9. フルテキスト検索の使用

#### 11.3.5.9.1. Fuzzy クエリー

ファジークエリー `add ~` を整数とともに実行するには、用語の後に使用される用語からの距離を表します。たとえば、以下ようになります。

```
FROM sample_bank_account.Transaction WHERE description : 'cofee'~2
```

#### 11.3.5.9.2. 範囲クエリー

以下の例に示すように、範囲クエリーを実行するには、中括弧のペア内で指定の境界を定義します。

```
FROM sample_bank_account.Transaction WHERE amount : [20 to 50]
```

#### 11.3.5.9.3. フレーズクエリー

次の例に示すように、単語のグループは引用符で囲むことで検索できます。

```
FROM sample_bank_account.Transaction WHERE description : 'bus fare'
```

#### 11.3.5.9.4. 近接クエリー

特定の距離内で2つの用語を検索して近接クエリーを実行するには、フレーズの後に距離とともに `~` を追加します。たとえば、以下の例では、キャンセルと `fee` という単語が3個以上ありません。

```
FROM sample_bank_account.Transaction WHERE description : 'canceling fee'~3
```

#### 11.3.5.9.5. ワイルドカードクエリー

"text" または "test" を検索するには、単一文字のワイルドカード検索 `?` を使用します。

```
FROM sample_bank_account.Transaction where description : 'te?t'
```

"test"、"tests"、"tester" を検索するには、マルチ文字のワイルドカード検索 `*` を使用します。

```
FROM sample_bank_account.Transaction where description : 'test*'
```

#### 11.3.5.9.6. 正規表現クエリー

正規表現クエリーは、`/` の間のパターンを指定することで実行できます。Ickle は Lucene の正規表現構文を使用しているため、単語 **moat** または **boat** を検索するには、以下を使用できます。

```
FROM sample_library.Book where title : /[mb]oat/
```

#### 11.3.5.9.7. クエリーのブースト

用語は、指定のクエリーにおける耐障害性を高めるために `^` を追加し、条件を強化できます。たとえば、ビールとビールとの関連性が 3 倍高いビールとワインを含むタイトルを検索するには、次のように使用できます。

```
FROM sample_library.Book WHERE title : beer^3 OR wine
```

## 11.4. 埋め込み検索

Data Grid をライブラリーとして使用すると、埋め込み検索を利用できます。protobuf マッピングは不要であり、インデックス作成と検索の両方が Java オブジェクト上で実行されます。

### 11.4.1. 簡単な例

books と呼ばれる Data Grid キャッシュに **Book** インスタンスを保存します。**Book** インスタンスはインデックス化されるため、キャッシュのインデックスを有効にします。

Data Grid の設定:

#### infinispan.xml

```
<infinispan>
  <cache-container>
    <transport cluster="infinispan-cluster"/>
    <distributed-cache name="books">
      <indexing>
        <indexed-entities>
          <indexed-entity>com.acme.Book</indexed-entity>
        </indexed-entities>
      </indexing>
    </distributed-cache>
  </cache-container>
</infinispan>
```

キャッシュを取得します。

```
import org.infinispan.Cache;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.manager.EmbeddedCacheManager;

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan.xml");
Cache<String, Book> cache = manager.getCache("books");
```

各 **Book** は、次の例のように定義されます。インデックスを作成するプロパティを選択する必要があります。プロパティごとに、Hibernate Search プロジェクトで定義されたアノテーションを使用して高度なインデックスオプションを任意で選択できます。

### Book.java

```
import org.hibernate.search.annotations.*;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

//Values you want to index need to be annotated with @Indexed, then you pick which fields and how
//they are to be indexed:
@Indexed
public class Book {
    @Field String title;
    @Field String description;
    @Field @DateBridge(resolution=Resolution.YEAR) Date publicationYear;
    @IndexedEmbedded Set<Author> authors = new HashSet<Author>();
}
```

### Author.java

```
public class Author {
    @Field String name;
    @Field String surname;
    // hashCode() and equals() omitted
}
```

Data Grid **Cache** に複数の **Book** インスタンスを保存したとすると、次の例のように、一致するフィールドを検索できます。

### QueryExample.java

```
// get the query factory from the cache:
QueryFactory queryFactory = org.infinispan.query.Search.getQueryFactory(cache);

// create an lckle query that will do a full-text search (operator ':') on fields 'title' and 'authors.name'
Query<Book> fullTextQuery = queryFactory.create("FROM com.acme.Book WHERE title:'infinispan'
AND authors.name:'sanne'")

// The ('=') operator is not a full-text operator, thus can be used in both indexed and non-indexed
// caches
Query<Book> exactMatchQuery = queryFactory.create("FROM com.acme.Book WHERE title =
'Programming Infinispan' AND authors.name = 'Sanne Grinovero'")

// Full-text and non-full text operators can be part of the same query
```

```

Query q = queryFactory.create("FROM com.query.Book b where b.author.name = 'Stephen' and
b.description : (+'dark' -'tower')");

// get the results
List<Book> found=query.execute().list();

```

**list()** とは別に、**iterator()** で取得するか、ページ付けを使用するかを選択できます。

## 11.4.2. マッピングエンティティ

Data Grid は、エンティティレベルでインデックス作成の詳細な設定を定義するため [Hibernate Search](#) の API に依存します。この設定には、アノテーションが付けられたフィールド、使用するアナライザー、ネストされたオブジェクトのマッピング方法などが含まれます。詳細なドキュメントは [the Hibernate Search manual](#) を参照してください。

### 11.4.2.1. @DocumentId

Hibernate Search とは異なり、**@DocumentId** を使用してフィールドを識別子としてマーク付けすると、Data Grid は値を保存するために使用されるキーになります。すべての **@Indexed** オブジェクトの識別子は、値を保存するために使用されるキーになります。**@Transformable**、カスタム型、およびカスタム **FieldBridge** 実装の組み合わせを使用して、キーのインデックス化方法をカスタマイズできます。

### 11.4.2.2. @Transformable keys

各値のキーはインデックス化する必要があり、キーインスタンスを **String** で変換する必要があります。Data Grid には、共通のプリミティブをエンコードするためのデフォルトの変換ルーチンが含まれていますが、カスタムキーを使用するには **org.infinispan.query.Transformer** の実装を提供する必要があります。

#### アノテーションを使用したキートランスフォーマーの登録

キークラスに **org.infinispan.query.Transformable** のアノテーションを付け、カスタムトランスフォーマー実装が自動的に選択されます。

```

@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
    ...
}

public class CustomTransformer implements Transformer {
    @Override
    public Object fromString(String s) {
        ...
        return new CustomKey(...);
    }

    @Override
    public String toString(Object customType) {
        CustomKey ck = (CustomKey) customType;
        return ...
    }
}

```

#### キャッシュインデックス設定を介したキートランスフォーマーの登録

埋め込みおよびサーバー設定の両方で、**key-transformers** xml 要素を使用します。

```
<replicated-cache name="test">
  <indexing auto-config="true">
    <key-transformers>
      <key-transformer key="com.mycompany.CustomKey"
transformer="com.mycompany.CustomTransformer"/>
    </key-transformers>
  </indexing>
</replicated-cache>
```

または、Java 設定 API(組み込みモード)を使用します。

```
ConfigurationBuilder builder = ...
builder.indexing().enable()
    .addKeyTransformer(CustomKey.class, CustomTransformer.class);
```

### 11.4.2.3. プログラムによるマッピング

アノテーションを使用してエンティティをインデックスにマップする代わりに、プログラムで設定することもできます。

次の例では、グリッドに格納され、クラスにアノテーションを付けなくても2つのプロパティで検索可能にするオブジェクト **Author** をマップします。

```
import org.apache.lucene.search.Query;
import org.hibernate.search.cfg.Environment;
import org.hibernate.search.cfg.SearchMapping;
import org.hibernate.search.query.dsl.QueryBuilder;
import org.infinispan.Cache;
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.configuration.cache.Index;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.query.CacheQuery;
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;

import java.io.IOException;
import java.lang.annotation.ElementType;
import java.util.Properties;

SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed()
    .property("name", ElementType.METHOD).field()
    .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(Environment.MODEL_MAPPING, mapping);
properties.put("hibernate.search.[other options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
    .indexing().index(Index.NONE)
    .withProperties(properties)
    .build();
```



```
DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "Manik", "Surtani");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q = qb.keyword().onField("name").matching("Manik").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
assert cq.getResultSize() == 1;
```

## 11.5. リモート検索

リモート検索は埋め込みと非常によく似ていますが、データはネットワークとストレージの両方のエンコーディングとして [Google Protocol Buffers](#) を使用する必要があるという顕著な違いがあります。さらに、Hibernate Search アノテーションに依存せずに、データ構造とインデックス要素を定義する protobuf スキーマを記述 (または Java クラスから生成) する必要があります。

protobuf を使用すると、リモートクエリーは Java だけでなく、REST、C#、および Node.js クライアントで機能します。

### 11.5.1. リモートクエリーの例

組み込みクエリーから Book Sample を再確認しますが、今回は Java Hot Rod クライアントと Infinispan サーバーを使用します。**Book** と呼ばれるオブジェクトは books と呼ばれる Infinispan キャッシュに保存されます。Book インスタンスはインデックス化されるため、キャッシュのインデックスを有効にします。

#### infinispan.xml

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="books">
      <indexing>
        <indexed-entities>
          <indexed-entity>book_sample.Book</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>
```

または、キャッシュのインデックス化がインデックス化されていない場合は、**<encoding>** を **application/x-protostream** として設定して、ストレージがクエリー可能であることを確認します。

#### infinispan.xml

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="books">
      <encoding media-type="application/x-protostream"/>
    </replicated-cache>
  </cache-container>
</infinispan>
```

```

</replicated-cache>
</cache-container>
</infinispan>

```

各 **Book** は以下の例のように定義されます。@Protofield アノテーションを使用してプロトコルバッファメッセージフィールドを識別し、フィールドの @ProtoDoc アノテーションを使用してインデックス属性を設定します。

### Book.java

```

import org.infinispan.protostream.annotations.ProtoDoc;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoDoc("@Indexed")
public class Book {

    @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
    @ProtoField(number = 1)
    final String title;

    @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
    @ProtoField(number = 2)
    final String description;

    @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
    @ProtoField(number = 3, defaultValue = "0")
    final int publicationYear;

    @ProtoFactory
    Book(String title, String description, int publicationYear) {
        this.title = title;
        this.description = description;
        this.publicationYear = publicationYear;
    }
    // public Getter methods omitted for brevity
}

```

上記のアノテーションは、コンパイル中に **Book** インスタンスの読み取り、書き込み、およびクエリーに必要なアーティファクトを生成します。この生成を有効にするには、空のコンストラクターまたはインターフェイスを使用して新しく作成されたクラスで @AutoProtoSchemaBuilder アノテーションを使用します。

### RemoteQueryInitializer.java

```

import org.infinispan.protostream.SerializationContextInitializer;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(
    includeClasses = {
        Book.class
    },
    schemaFileName = "book.proto",
    schemaFilePath = "proto/",

```

```

    schemaPackageName = "book_sample")
    public interface RemoteQueryInitializer extends SerializationContextInitializer {
    }

```

コンパイル後、ファイル **book.proto** ファイルが、アノテーションが付いたインターフェイスの実装 **RemoteQueryInitializerImpl.java** とともに、設定された **schemaFilePath** に作成されます。この具体的なクラスは Hot Rod クライアントコードで直接使用して、シリアル化コンテキストを初期化します。

すべてを1つにまとめます。

## RemoteQuery.java

```

package org.infinispan;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.Search;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.remote.client.ProtobufMetadataManagerConstants;

public class RemoteQuery {

    public static void main(String[] args) throws Exception {
        ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
        // RemoteQueryInitializerImpl is generated
        clientBuilder.addServer().host("127.0.0.1").port(11222)
            .security().authentication().username("user").password("user")
            .addContextInitializers(new RemoteQueryInitializerImpl());

        RemoteCacheManager remoteCacheManager = new
        RemoteCacheManager(clientBuilder.build());

        // Grab the generated protobuf schema and registers in the server.
        Path proto = Paths.get(RemoteQuery.class.getClassLoader()
            .getResource("proto/book.proto").toURI());
        String protoBufCacheName =
        ProtobufMetadataManagerConstants.PROTOBUF_METADATA_CACHE_NAME;
        remoteCacheManager.getCache(protoBufCacheName).put("book.proto", Files.readString(proto));

        // Obtain the 'books' remote cache
        RemoteCache<Object, Object> remoteCache = remoteCacheManager.getCache("books");

        // Add some Books
        Book book1 = new Book("Infinispan in Action", "Learn Infinispan with using it", 2015);
        Book book2 = new Book("Cloud-Native Applications with Java and Quarkus", "Build robust and
        reliable cloud applications", 2019);

        remoteCache.put(1, book1);
        remoteCache.put(2, book2);
    }
}

```

```
// Execute a full-text query
QueryFactory queryFactory = Search.getQueryFactory(remoteCache);
Query<Book> query = queryFactory.create("FROM book_sample.Book WHERE title:'java'");

List<Book> list = query.execute().list(); // Voila! We have our book back from the cache!
}
}
```

## 11.5.2. Protobuf でエンコードされたエントリーのインデックス化

[Remote Query の例](#)にあるように、protobuf エンティティのクエリーに必要な1つの手順は、クライアントおよびサーバーにエンティティに関連するメタデータ (.proto ファイル) を提供することです。

記述子は `__protobuf_metadata` という名前のサーバーで専用キャッシュに保存されます。このキャッシュのキーと値はどちらもプレーンテキストの文字列です。したがって、新しいスキーマの登録は、スキーマ名をキーとして、スキーマファイル自体を値として使用して、このキャッシュに対して `put()` 操作を実行するのと同じくらい簡単です。

代わりに、CLI(`cache-container=*:register-proto-schemas()` 操作経由)、管理コンソール、REST エンドポイント `/rest/v2/schemas`、または JMX 経由で `ProtobufMetadataManager` MBean を使用できます。セキュリティが有効になっている場合、リモートプロトコル経由でスキーマキャッシュにアクセスするには、ユーザーが `'__schema_manager'` ロールに属している必要があります。



### 注記

キャッシュに対してインデックスが有効になっている場合でも、protobuf スキーマドキュメントアノテーション (`@ProtoDoc`) 内の `@Indexed` および `@Field` を使用してインデックスする必要のあるフィールドを指定しない限り、Protobuf でエンコードされたエントリーのフィールドはインデックス付けされません。

## 11.5.3. 分析

分析は、入力データを、インデックスを作成してクエリーできる1つ以上の用語に変換するプロセスです。[Embedded Query](#) のマッピングは、Lucene ベースのアナライザーのセットをサポートする [Hibernate Search アノテーション](#) を介して行われますが、クライアントサーバーモードでは、アナライザー定義はプラットフォームに依存しない方法で宣言されます。

### 11.5.3.1. デフォルトのアナライザー

Data Grid は、以下のようにリモートクエリーに対してデフォルトのアナライザーのセットを提供します。

定義	説明
<b>standard</b>	テキストフィールドをトークンに分割し、空白と句読点を区切り文字として扱います。
<b>simple</b>	非文字で区切り、すべての文字を小文字に変換することにより、入力ストリームをトークン化します。空白と非文字は破棄されます。

定義	説明
<b>whitespace</b>	テキストストリームを空白で分割し、空白以外の文字のシーケンスをトークンとして返します。
<b>キーワード</b>	テキストフィールド全体を単一トークンとして扱います。
<b>stemmer</b>	SnowballPorter フィルターを使用して英語の単語を語幹にします。
<b>ngram</b>	デフォルトでサイズ3つのグラムである n-gram トークンを生成します。
<b>filename</b>	テキストフィールドを <b>standard</b> アナライザーよりも大きなサイズトークンに分割し、空白文字を区切り文字として扱い、すべての文字を小文字に変換します。

これらのアナライザー定義は Apache Lucene をベースとし、as-is で提供されます。tokenizers、filters、および CharFilters に関する詳細は、適切な Lucene のドキュメントを参照してください。

### 11.5.3.2. アナライザー定義の使用

アナライザー定義を使用するには、**.proto** スキーマファイルで名前ですれらを参照します。

1. **Analyze.YES** 属性を追加して、プロパティーが分析されていることを示します。
2. **@Analyzer** アノテーションでアナライザー定義を指定します。

以下は、参照されたアナライザー定義の例になります。

```
/* @Indexed */
message TestEntity {

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer = @Analyzer(definition =
"keyword")) */
    optional string id = 1;

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer = @Analyzer(definition = "simple"))
*/
    optional string name = 2;
}
```

**@ProtoField** アノテーションが付けられた Java クラスを使用する場合、宣言は以下のようになります。

```
@ProtoDoc("@Field(store = Store.YES, analyze = Analyze.YES, analyzer = @Analyzer(definition =
\"keyword\"))")
@ProtoField(number = 1)
```

```
final String id;
```

```
@ProtoDoc("@Field(store = Store.YES, analyze = Analyze.YES, analyzer = @Analyzer(definition = \"simple\"))")
```

```
@ProtoField(number = 2)
```

```
final String description;
```

### 11.5.3.3. カスタムアナライザー定義の作成

カスタムアナライザー定義が必要な場合は、以下を行います。

1. **JAR** ファイルにパッケージ化された **ProgrammaticSearchMappingProvider** インターフェイスの実装を作成します。
2. **JAR** の **META-INF/services/** ディレクトリーに **org.infinispan.query.spi.ProgrammaticSearchMappingProvider** という名前のファイルを設定します。このファイルには、実装の完全修飾クラス名が含まれている必要があります。
3. **JAR** を Data Grid インストールの **lib/** ディレクトリーにコピーします。



#### 重要

jar は、起動時に Data Grid サーバーで利用できる必要があります。サーバーがすでに実行中の場合は、追加できません。

以下は、**ProgrammaticSearchMappingProvider** インターフェイスの実装例です。

```
import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.core.StopFilterFactory;
import org.apache.lucene.analysis.standard.StandardFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;
import org.hibernate.search.cfg.SearchMapping;
import org.infinispan.Cache;
import org.infinispan.query.spi.ProgrammaticSearchMappingProvider;

public final class MyAnalyzerProvider implements ProgrammaticSearchMappingProvider {

    @Override
    public void defineMappings(Cache cache, SearchMapping searchMapping) {
        searchMapping
            .analyzerDef("standard-with-stop", StandardTokenizerFactory.class)
            .filter(StandardFilterFactory.class)
            .filter(LowerCaseFilterFactory.class)
            .filter(StopFilterFactory.class);
    }
}
```

## 11.6. 継続的なクエリー

継続的なクエリーにより、アプリケーションはクエリーフィルターに現在一致したエントリーを受信するリスナーを登録し、さらにキャッシュ操作の結果としてクエリーされたデータセットへの変更を継続的に通知できます。これには、セットに結合された値の着信一致、更新された一致、変更されて引き続き一致する一致値、およびセットを離れた値の発信一致が含まれます。継続的なクエリーを使用するこ

とにより、アプリケーションは、変更を検出するために同じクエリーを繰り返し実行する代わりに、イベントの安定したストリームを受信し、リソースがより効率的に使用されるようになります。たとえば、以下のユースケースすべてで、継続的なクエリーを使用できます。

- Person エンティティーに **age** プロパティーがあり、ユーザーアプリケーションによって更新される 18~25 歳の人を返す。
- \$2000 を超えるすべてのトランザクションを返す。
- F1 レーサーのラップスピードが 1:45.00 秒未満だったすべての時間を返す (キャッシュにラップエントリーが含まれていて、レース中にラップがライブ入力されていると仮定)。

### 11.6.1. 連続クエリー実行

継続的なクエリーは、以下の場合に通知されるリスナーを使用します。

- エントリーは、**Join** イベントによって表される指定のクエリーの一致を開始します。
- 一致するエントリーが更新され、**Update** イベントによって表されるクエリーの一致が継続されません。
- エントリーは、**Leave** イベントで表されるクエリーの一致を停止します。

クライアントが継続的なクエリーリスナーを登録すると、すぐにクエリーに一致する結果の受信を開始します (上記のように **Join** イベントとして受信された)。さらに、通常は作成、変更、削除、または有効期限イベントを生成するキャッシュ操作の結果として、他のエントリーが **Join** イベントとしてクエリーの一致を開始したとき、または **Leave** イベントとしてクエリーの一致を停止したときに、後続の通知を受信します。更新されたキャッシュエントリーは、操作の前後でエントリーがクエリーフィルターに一致する場合、**Update** イベントを生成します。要約すると、リスナーが **Join**、**Update**、または **Leave** イベントを受信するかどうかを決定するために使用されるロジックは次のとおりです。

1. 古い値と新しい値の両方に対するクエリーが false と評価された場合、イベントは抑制されません。
2. 古い値に対するクエリーが false と評価され、新しい値に対するクエリーが true と評価された場合、**Join** イベントが送信されます。
3. 古い値と新しい値の両方のクエリーが true と評価されると、**Update** イベントが送信されません。
4. 古い値に対するクエリーが true と評価され、新しい値に対するクエリーが false と評価された場合、**Leave** イベントが送信されます。
5. 古い値に対するクエリーが true と評価され、エントリーが削除または期限切れになると、**Leave** イベントが送信されます。



#### 注記

継続的なクエリーは、グループ化、集約、およびソート操作以外のすべてのクエリー機能を使用できます。

### 11.6.2. 継続的なクエリーの実行

継続的なクエリーを作成するには、以下を実行します。

1. クエリーオブジェクトを作成します。[検索セクション](#)を参照してください。



2. 適切なメソッドを呼び出して、キャッシュの `ContinuousQuery`(`org.infinispan.query.api.continuous.ContinuousQuery` オブジェクト) を取得します。
  - リモートモードの `org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)`
  - 組み込みモードの `org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)`
3. 以下のように、クエリーと継続的なクエリーリスナー (`org.infinispan.query.api.continuous.ContinuousQueryListener`) を登録します。

```
continuousQuery.addContinuousQueryListener(query, listener);
```

以下の例は、埋め込みモードの単純な継続的なクエリーのユースケースを示しています。

### 継続的クエリーの登録

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

[...]

// We have a cache of Persons
Cache<Integer, Person> cache = ...

// We begin by creating a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery = Search.getContinuousQuery(cache);

// Define our query. In this case we will be looking for any Person instances under 21 years of age.
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.create("FROM Person p WHERE p.age < 21");

final Map<Integer, Person> matches = new ConcurrentHashMap<Integer, Person>();

// Define the ContinuousQueryListener
ContinuousQueryListener<Integer, Person> listener = new ContinuousQueryListener<Integer,
Person>() {
    @Override
    public void resultJoining(Integer key, Person value) {
        matches.put(key, value);
    }

    @Override
    public void resultUpdated(Integer key, Person value) {
        // we do not process this event
    }

    @Override
```



```

public void resultLeaving(Integer key) {
    matches.remove(key);
}
};

// Add the listener and the query
continuousQuery.addContinuousQueryListener(query, listener);

[...]

// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);

```

21歳未満の Person インスタンスがキャッシュに追加されると、リスナーによって受信され、**matches** マップに配置されます。これらのエントリがキャッシュから削除されるか、年齢が 21 歳以上に変更されると、それらは **matches** から削除されます。

### 11.6.3. 継続的なクエリーの削除

クエリーのそれ以上の実行を停止するには、リスナーを単に削除します。

```
continuousQuery.removeContinuousQueryListener(listener);
```

### 11.6.4. 継続的なクエリーのパフォーマンスに関する注意

継続的なクエリーは、アプリケーションに一定の更新ストリームを提供するように設計されており、特に幅広いクエリーに対して非常に多くのイベントが生成される可能性があります。イベントごとに新規の一時的なメモリ割り当てが行われます。この動作によりメモリが不足し、クエリーが適切に設計されていない場合、**OutOfMemoryErrors**(特にリモートモード)が発生する可能性があります。このような問題を防ぐために、一致するエントリーの数と各一致のサイズの両方の観点から、各クエリーが必要な最小限の情報をキャプチャーし(プロジェクションを使用して興味深いプロパティをキャプチャできます)、各 **ContinuousQueryListener** が受信したすべてのイベントをブロックせずにすばやく処理し、リッスンするキャッシュから一致する新しいイベントの生成につながるアクションの実行を回避するように設計されていることが強く推奨されます。

## 11.7. 統計

以下のコードスニペットが示すように、クエリー [統計](#) は **SearchManager** から取得できます。

```

SearchManager searchManager = Search.getSearchManager(cache);
org.hibernate.search.stat.Statistics statistics = searchManager.getStatistics();

```

### ヒント

このデータは、**org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=Statistics** の名前で登録された [Hibernate Search StatisticsInfoMBean](#) を介して JMX 経由でも利用できます。この MBean は常に Data Grid によって登録されますが、統計はキャッシュレベルで統計収集が有効になっている場合にのみ収集されることに注意してください。



## 警告

Hibernate Search には、[ここで説明するように](#)、JMX 統計の独自の設定プロパティ `hibernate.search.jmx_enabled` および `hibernate.search.generate_statistics` があります。Data Grid Query でそれらを使用することは、重複した MBean や予測不能な結果のみが発生するため、禁止されています。

## 11.8. パフォーマンスチューニング

### 11.8.1. SYNC モードでのバッチ書き込み

デフォルトでは、[Index Managers](#) は sync モードで動作します。つまり、データが Data Grid に書き込まれると、インデックス操作が同期的に実行されます。この同期性により、インデックスは常にデータと整合性が保たれます (したがって、検索で表示されます) が、インデックスへのコミットも実行されるため、書き込み操作が遅くなる可能性があります。Lucene ではコミットは非常にコストのかかる操作であるため、異なるノードからの複数の書き込みを自動的に1つのコミットにバッチ処理して、影響を軽減できます。

したがって、インデックスを有効にして Data Grid にデータを読み込む場合は、複数のスレッドを使用してこのバッチ処理を活用してみてください。

複数のスレッドを使用しても必要なパフォーマンスが得られない場合は、インデックスを一時的に無効にしてデータをロードし、後で [再インデックス化](#) 操作を実行することもできます。これは、`SKIP_INDEXING` フラグを使用してデータを書き込むことで実行できます。

```
cache.getAdvancedCache().withFlags(Flag.SKIP_INDEXING).put("key","value");
```

### 11.8.2. 非同期モードを使用した書き込み

データ書き込み間のわずかな遅延が許容可能であり、そのデータがクエリーに表示される場合は、インデックスマネージャーを非同期モードで動作するように設定できます。非同期モードでは、設定可能な間隔でコミットが行われるため、書き込みパフォーマンスが大幅に向上します。

設定:

```
<distributed-cache name="default">
  <indexing>
    <!-- Index data in async mode -->
    <property name="default.worker.execution">async</property>
    <!-- Optional: configure the commit interval, default is 1000ms -->
    <property name="default.index_flush_interval">500</property>
  </indexing>
</distributed-cache>
```

### 11.8.3. インデックスリーダーの非同期ストラテジー

Lucene は、内部的にインデックスのスナップショットと連携します。`IndexReader` が開かれると、開いた時点までのインデックスの変更のみが表示されます。`IndexReader` が更新されるまでさらにイン

デックスの変更は表示されません。Data Grid でデフォルトで使用されるインデックスマネージャーは、すべてのクエリーの前にインデックスリーダーが更新されているかをチェックし、必要に応じてそれらを更新します。

**async** として設定された **reader.strategy** 設定を使用すると、このストラテジーを調整して、この更新チェックを事前設定された間隔に緩和することができます。

```
<distributed-cache name="default">
  <indexing>
    <property name="default.reader.strategy">async</property>
    <!-- refresh reader every 1s, default is 5s -->
    <property name="default.reader.async_refresh_period_ms">1000</property>
  </indexing>
</distributed-cache>
```

#### 11.8.4. Lucene オプション

Lucene でチューニングオプションを直接適用することが可能です。詳細は、[Hibernate Search マニュアル](#) を参照してください。

## 第12章 GRID でのコードの実行

キャッシュの主な利点は、マシン全体でもキーで値を迅速に検索できることです。実際、この理由だけで、おそらく多くのユーザーが Data Grid を使用しています。ただし、Data Grid には、すぐには明らかにならない多くの利点があります。通常、Data Grid はマシンのクラスターで使用されるため、ユーザーのニーズのワークロードを実行するためにクラスター全体を利用するのに役立つ機能もあります。



### 注記

このセクションでは、埋め込みキャッシュを使用したグリッドでのコードの実行についてのみ説明します。リモートキャッシュを使用している場合は、リモートグリッドでのコードの実行に関する詳細を確認する必要があります。

### 12.1. クラスターエグゼキューター

マシンのグループがあるため、それらすべてでコードを実行するためにそれらの結合された計算能力を活用することは理にかなっています。キャッシュマネージャーには、クラスター内で任意のコードを実行できる優れたユーティリティが付属しています。この機能にはキャッシュを使用する必要はありません。この **クラスターエグゼキューター** は、**EmbeddedCacheManager** で `executor()` を呼び出すことで取得できます。このエグゼキューターは、クラスター設定と非クラスター設定の両方で取得できます。



### 注記

`ClusterExecutor` は、コードがキャッシュ内のデータに依存しないコードを実行するために特別に設計されており、代わりに、ユーザーがクラスター内でコードを簡単に実行できるようにする方法として使用されます。

このマネージャーは、Java 8 を使用して特別に構築されており、機能的な API を念頭に置いているため、すべてのメソッドは機能的なインターフェイスを引数として取ります。また、これらの引数は他のノードに送信されるため、シリアライズする必要があります。ラムダがすぐに `Serializable` になるような策を使用しています。つまり、引数に `Serializable` と実際の引数タイプ (つまり、`Runnable` または `Function`) の両方を実装させることです。JRE は、呼び出す方法を決定する際に最も具体的なクラスを選択するため、ラムダは常にシリアライズ可能です。また、`Externalizer` を使用してメッセージサイズをさらに減らすこともできます。

マネージャーはデフォルトで、指定されたコマンドを、送信元のノードを含むクラスター内のすべてのノードに送信します。セクションで説明されているように、`filterTargets` メソッドを使用して、タスクが実行するノードを制御できます。

#### 12.1.1. 実行ノードのフィルタリング

コマンドを実行するノードを制限できます。たとえば、同じラック内のマシンでのみ計算を実行したい場合があります。または、ローカルサイトで1回、別のサイトで操作を再実行することもできます。クラスターエグゼキューターは、同じマシン、ラック、またはサイトレベルの範囲で要求を送信するノードを制限できます。

#### SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

このトポロジーベースフィルタリングを使用するには、サーバーヒントを介してトポロジー対応のコンシステントハッシュを有効にする必要があります。

ノードの **Address** に基づいて述部を使用してフィルタリングすることもできます。これは任意で、以前のコードスニペットでトポロジベースのフィルタリングと組み合わせることもできます。

また、実行対象と見なすことができるノードを除外する **Predicate** を使用して、任意の方法でターゲットノードを選択することもできます。これは同時に Topology フィルタリングと組み合わせて、クラスター内でコードを実行する場所をより詳細に制御できるようにすることもできます。

### Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a -> a.equals(..)).submit(...)
```

### 12.1.2. Timeout

クラスターエグゼキューターを使用すると、呼び出しごとにタイムアウトを設定できます。デフォルトは、Transport Configuration で設定された分散同期のタイムアウトになります。このタイムアウトは、クラスター化されたキャッシュマネージャーとクラスター化されていないキャッシュマネージャーの両方で機能します。タイムアウトの期限が切れると、エグゼキューターがタスクを実行しているスレッドを中断する場合と中断しない場合があります。ただし、タイムアウトが発生すると、**Consumer** または **Future** は **TimeoutException** を渡して完了します。この値は、`timeout` メソッドを呼び出して、希望の期間を指定することでオーバーライドすることができます。

### 12.1.3. 単一ノードの提出

クラスターエグゼキューターは、すべてのノードにコマンドを送信する代わりに、単一ノード送信モードで実行することもできます。代わりに、通常はコマンドを受信するノードの1つを選択し、1つだけに送信します。それぞれの送信は、別のノードを使用してタスクが実行される可能性があります。これは、ClusterExecutor が実装する **java.util.concurrent.Executor** として ClusterExecutor を使用するのが非常に便利です。

### SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

#### 12.1.3.1. Failover

シングルノード送信で実行する場合は、コマンドを再試行することにより、特定のコマンドの処理中に例外が発生した場合にクラスターエグゼキューターが処理できるようにすることが望ましい場合があります。これが発生すると、クラスターエグゼキューターは単一のノードを再度選択し、任意のフェイルオーバー試行までコマンドを再実行します。選択したノードは、トポロジーまたは述部のチェックをパスするノードである可能性があることに注意してください。フェイルオーバーは、オーバーライドされた `singleNodeSubmission` メソッドを呼び出すことで有効になります。指定されたコマンドは、コマンドが例外なく完了するか、送信の合計量が指定されたフェイルオーバーカウントと等しくなるまで、単一のノードに再送信されます。

### 12.1.4. 以下に例を示します。PI 近似

この例は、ClusterExecutor を使用して PI の値を見積もる方法を示しています。

Pi 近似は、クラスターエグゼキューターを介した並列分散実行から大きな利点を得ることができます。

正方形の面積は  $S_a = 4r^2$  であり、円の面積は  $C_a = \pi r^2$  であることを思い出してください。2つ目の式からの  $r^2$  を置き換えると、 $\pi = 4 * C_a / S_a$  になります。ここで、正方形に非常に多くのダーツを射ることができると仮定して、射ったダーツの総数に対して円の中に入ったダーツの割合を取ると、 $C_a / S_a$  の値が近似します。 $\pi = 4 * C_a / S_a$  であるため、 $\pi$  の近似値を簡単に導き出すことができます。ダーツを多く撃つほど、より良い近似が得られます。以下の例では、10億本のダーツを撃ちますが、それらを連続して撃つのではなく、Data Grid クラスター全体でダーツ射撃の作業を並列化します。これは1のクラスターで正常に機能しますが、遅くなることに注意してください。

```
public class PiAppx {

    public static void main (String [] arg){
        EmbeddedCacheManager cacheManager = ..
        boolean isCluster = ..

        int numPoints = 1_000_000_000;
        int numServers = isCluster ? cacheManager.getMembers().size() : 1;
        int numberPerWorker = numPoints / numServers;

        ClusterExecutor clusterExecutor = cacheManager.executor();
        long start = System.currentTimeMillis();
        // We receive results concurrently - need to handle that
        AtomicLong countCircle = new AtomicLong();
        CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
            int insideCircleCount = 0;
            for (int i = 0; i < numberPerWorker; i++) {
                double x = Math.random();
                double y = Math.random();
                if (insideCircle(x, y))
                    insideCircleCount++;
            }
            return insideCircleCount;
        }, (address, count, throwable) -> {
            if (throwable != null) {
                throwable.printStackTrace();
                System.out.println("Address: " + address + " encountered an error: " + throwable);
            } else {
                countCircle.getAndAdd(count);
            }
        });
        fut.whenComplete((v, t) -> {
            // This is invoked after all nodes have responded with a value or exception
            if (t != null) {
                t.printStackTrace();
                System.out.println("Exception encountered while waiting:" + t);
            } else {
                double appxPi = 4.0 * countCircle.get() / numPoints;

                System.out.println("Distributed PI appx is " + appxPi +
                    " using " + numServers + " node(s), completed in " + (System.currentTimeMillis() - start) +
                    " ms");
            }
        });

        // May have to sleep here to keep alive if no user threads left
    }
}
```

```
private static boolean insideCircle(double x, double y) {  
    return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))  
        <= Math.pow(0.5, 2);  
}  
}
```



## 第13章 ストリーム

結果を生成するために、キャッシュ内のサブセットまたはすべてのデータを処理したい場合があります。これにより、マップの削減が可能になります。Data Gridを使用すると、ユーザーは非常によく似た操作を実行できますが、標準の JRE API を使用して実行できます。Java 8 では、ユーザーがデータに対して処理を細かく反復するのではなく、コレクションで機能スタイルの操作を可能にする **ストリーム** の概念が導入されました。ストリーム操作は、MapReduce と似た方法で実装できます。MapReduce と同様、キャッシュ全体で処理を実行できますが、非常に大きなデータセットになりますが、効率的な方法になります。



### 注記

ストリームは、クラスタートポロジの変更自動的に調整されるため、キャッシュに存在するデータを扱う場合に推奨される方法です。

また、エントリーの反復方法を制御できるため、クラスター全体ですべての操作を同時に実行する場合は、分散されたキャッシュで操作をより効率的に実行できます。

ストリームは、`stream` メソッドまたは `parallelStream` メソッドを呼び出して、Cache から返される `entrySet`、`keySet`、または `values` コレクションから取得されます。

### 13.1. 一般的なストリーム操作

本セクションでは、使用している基礎となるキャッシュの種類に関係なく、さまざまなオプションを説明します。

### 13.2. キーのフィルタリング

特定のキーのサブセットでのみ動作するようにストリームをフィルターできます。これは、**CacheStream** で `filterKeys` メソッドを呼び出して実行できます。これは常に述部 **フィルター** で使用する必要があります。述部がすべてのキーを保持する場合はより高速になります。

**AdvancedCache** インターフェイスに慣れている人なら、なぜこの `keyFilter` ではなく `getAll` を使うのか不思議に思うかもしれません。エントリーをそのまま必要とし、それらすべてをローカルノードのメモリに必要とする場合、`getAll` を使用することにはいくつかの小さな利点 (ほとんどの場合ペイロードが小さい) があります。ただし、これらの要素で処理を行う必要がある場合は、分散並列処理とスレッド並列処理の両方を無料で取得できるため、ストリームを推奨します。

### 13.3. セグメントベースのフィルタリング



### 注記

これは高度な機能で、Data Grid セグメントおよびハッシュ技術の深い知識でのみ使用する必要があります。これらのセグメントベースのフィルタリングは、データを個別の呼び出しに分割する必要がある場合に便利です。これは、**Apache Spark** などの他のツールと統合する際に便利です。

このオプションは、レプリケートされたキャッシュと分散されたキャッシュでのみサポートされます。これにより、ユーザーは **KeyPartitioner** によって決定されるタイミングで、データのサブセットで操作することができます。このセグメントは、**CacheStream** で `filterKeySegments` メソッドを呼び出してフィルタリングできます。これは、キーフィルターの後に、中間操作が実行される前に適用されます。



## 13.4. ローカル/無効化

ローカルキャッシュまたは無効化キャッシュで使用するストリームは、通常のコレクションでストリームを使用する場合とまったく同じように使用できます。Data Grid は、必要に応じてすべての変換をバックグラウンドで処理し、より興味深いすべてのオプション (つまり `storeAsBinary` およびキャッシュローダー) で機能します。ストリーム操作が実行されるノードにローカルデータのみが使用されます。たとえば、無効化はローカルエントリーのみを使用します。

## 13.5. 例

以下のコードはキャッシュを取得し、値に "JBoss" の文字列が含まれるすべてのキャッシュエントリーを持つマップを返します。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(e -> e.getValue().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

## 13.6. 配布/複製/散在

これは、ストリームがストライドになるところです。ストリーム操作が実行されると、関連データを持つ各ノードにさまざまな中間操作と端末操作が送信されます。これにより、データを所有するノードで中間値を処理し、最終結果を元のノードにのみ送信し、パフォーマンスが向上します。

### 13.6.1. 再ハッシュ対応

内部的にはデータがセグメント化され、各ノードはプライマリー所有者として所有するデータでのみ操作を実行します。これにより、セグメントが各ノードで等量のデータを提供するのに十分な粒度であると仮定して、データを均等に処理できます。

分散キャッシュを使用する場合には、新規ノードが加わったり、残ったりすると、データをノード間で再シャッフルすることができます。分散ストリームはこのデータの再シャッフルを自動的に処理するため、ノードがクラスターを離れたり、クラスターに参加したりするときの監視について心配する必要はありません。シャッフルされたエントリーは 2 回処理される可能性があり、重複処理の量を制限するために、キーレベルまたはセグメントレベル (端末操作に応じて) で処理されたエントリーを追跡します。

ストリームで再ハッシュ認識を無効にすることは可能ですが、推奨されません。これは、再ハッシュが発生したときに、リクエストがデータのサブセットの確認を処理できる場合に限り考慮する必要があります。これは、`CacheStream.disableRehashAware()` を呼び出すことで実行できます。再ハッシュが発生しない場合、ほとんどの操作のパフォーマンスの向上は、完全に無視できます。唯一の例外は、処理されたキーを追跡する必要がないため、使用するメモリーが少ない `iterator` と `forEach` です。



#### 警告

自分が何をしているかを本当に理解していない限り、再ハッシュ認識を無効にすることを再考してください。

### 13.6.2. シリアル化

操作は他のノード全体に送信されるため、Data Grid マーシャリングでシリアル化できる必要があります。これにより、他のノードに操作を送信できます。

最も簡単な方法は、CacheStream インスタンスを使用し、通常どおりラムダを使用することです。Data Grid は、さまざまな Stream 中間メソッドおよび端末メソッドをすべて上書きして、引数の Serializable バージョン (SerializableFunction、SerializablePredicate など) を取得します。) でこれらのメソッドを [CacheStream](#) で見つけることができます。これは、[ここ](#) で定義されている最も具体的な方法を選択するための仕様に依存しています。

上記の例では、**Collector** を使用してすべての結果を **Map** に収集しました。ただし、**Collector** クラスは Serializable インスタンスを生成しません。そのため、これらを使用する必要がある場合は、2つの方法があります。

1つのオプションとして、**Supplier<Collector>** の指定を可能にする [CacheCollectors](#) クラスを使用します。このインスタンスは、シリアル化されていない **Collector** を提供するために、[Collectors](#) を使用することができます。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(e -> e.getValue().contains("Jboss"))
    .collect(CacheCollectors.serializableCollector() -> Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));
```

または、[CacheCollectors](#) の使用を回避し、代わりに **Supplier<Collector>** を取得するオーバーロードされた **collect** メソッドを使用できます。オーバーロードされた **collect** メソッドは **CacheStream** インターフェイスでしか利用できません。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(e -> e.getValue().contains("Jboss"))
    .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

ただし、**Cache** および **CacheStream** インターフェイスを使用できない場合は、**Serializable** 引数を使用できないため、ラムダを複数インターフェイスをキャストすることで、ラムダを **Serializable** に手動でキャストする必要があります。優れた方法ではありませんが、設定することは可能です。

```
Map<Object, String> jbossValues = map.entrySet().stream()
    .filter((Serializable & Predicate<Map.Entry<Object, String>>) e ->
e.getValue().contains("Jboss"))
    .collect(CacheCollectors.serializableCollector() -> Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));
```

推奨される最も高性能な方法は、最小限のペイロードを提供するために、**AdvancedExternalizer** を使用することです。残念ながら、これは、高度なエクスターナライザーが事前にクラスを定義する必要があるため、ラムダを使用できないことを意味します。

以下に示すように、高度なエクスターナライザーを使用できます。

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(new ContainsFilter("Jboss"))
    .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

```
class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
    private final String target;
```

```
    ContainsFilter(String target) {
        this.target = target;
```

```

    }

    @Override
    public boolean test(Map.Entry<Object, String> e) {
        return e.getValue().contains(target);
    }
}

class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

    @Override
    public Set<Class<? extends ContainsFilter>> getTypeClasses() {
        return Util.asSet(ContainsFilter.class);
    }

    @Override
    public Integer getId() {
        return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ContainsFilter object) throws IOException {
        output.writeUTF(object.target);
    }

    @Override
    public ContainsFilter readObject(ObjectInput input) throws IOException,
    ClassNotFoundException {
        return new ContainsFilter(input.readUTF());
    }
}

```

コレクターサプライヤーに高度なエクスターナライザーを使用して、ペイロードサイズをさらに減らすこともできます。

```

Map<Object, String> map = (Map<Object, String>) cache.entrySet().stream()
    .filter(new ContainsFilter("Jboss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?, Map<K,
U>>> {
    static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

    private ToMapCollectorSupplier() { }

    @Override
    public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
        return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
    }
}

class ToMapCollectorSupplierExternalizer implements
AdvancedExternalizer<ToMapCollectorSupplier> {

    @Override
    public Set<Class<? extends ToMapCollectorSupplier>> getTypeClasses() {

```

```

        return Util.asSet(ToMapCollectorSupplier.class);
    }

    @Override
    public Integer getId() {
        return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ToMapCollectorSupplier object) throws IOException
    {
    }

    @Override
    public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
    ClassNotFoundException {
        return ToMapCollectorSupplier.INSTANCE;
    }
}

```

## 13.7. 並列計算

分散ストリームは、デフォルトではできるだけ並列処理を試みます。エンドユーザーはこれを制御でき、実際にはオプションのいずれかを制御する必要があります。これらのストリームを並列化する方法は2つあります。

**各ノードにローカル** キャッシュコレクションからストリームを作成している場合、エンドユーザーは `stream` または `parallelStream` メソッドの呼び出しのいずれかを選択できます。並列ストリームが選択されたかどうかに応じて、各ノードに対してローカルで複数のスレッドが有効になります。再ハッシュ対応の `iterator` や `forEach` オペレーションなどの一部のオペレーションは、常にローカルで順次ストリームを使用することに注意してください。これは、並行ストリームをローカルに許可するように、ある時点で強化できます。

ローカルの並列処理を使用する場合は、計算が高速にかかる多数のエントリや操作が必要になるため注意が必要です。また、ユーザーが `forEach` で並列ストリームを使用する場合、これは通常は計算オペレーションに予約されている共有プールで実行されるため、アクションをブロックしないようにする必要があります。

**リモートリクエスト** 複数のノードがある場合に、リモート要求をすべて同時に処理するか、一度に1つずつ処理するかを制御することが望ましい場合があります。デフォルトでは、`iterator` 以外のすべての端末オペレーションは同時リクエストを実行します。`iterator` は、ローカルノードでのメモリー使用量全体を減らす方法であり、実際に実行する連続要求のみを実行します。

ユーザーがこのデフォルトを変更したい場合は、**CacheStream** で `sequentialDistribution` または `parallelDistribution` メソッドを呼び出して実行できます。

## 13.8. タスクのタイムアウト

操作リクエストのタイムアウト値を設定できます。このタイムアウトはリモートリクエストのタイムアウトにのみ使用され、リクエストごとに使用されます。前者はローカル実行がタイムアウトしないことを意味し、後者は上記のようなフェイルオーバーシナリオがある場合、後続のリクエストにはそれぞれ新しいタイムアウトがあることを意味します。タイムアウトを指定しないと、レプリケーションのタイムアウトをデフォルトのタイムアウトとして使用します。以下を実行することで、タスクでタイムアウトを設定できます。

```
CacheStream<Map.Entry<Object, String>> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

詳細は、java ドキュメントの [timeout](#) を確認してください。

## 13.9. 注入

`Stream` には、`forEach` と呼ばれる端末オペレーションがあり、データに副次的な影響を与える操作を実行できます。この場合、このストリームをサポートする `Cache` への参照を取得することが推奨されます。`Consumer` が `CacheAware` インターフェイスを実装する場合は、`Consumer` インターフェイスからの `accept` メソッドの前に `injectCache` メソッドが呼び出されます。

## 13.10. 分散ストリームの実行

分散ストリームの実行は、マップの削減に非常に似ています。ここでは、ゼロを多数の中間操作 (マップ、フィルターなど) に送信し、1つの端末オペレーションが各種ノードに送信します。オペレーションは、基本的に次のようになります。

1. 必要なセグメントは、どのノードが指定のセグメントのプライマリ所有者であるかによってグループ化されます。
2. リクエストが生成され、処理すべきセグメントを含む中間および端末オペレーションが含まれる各リモートノードに送信されます。
  - a. 端末オペレーションは、必要に応じてローカルで実行されます。
  - b. 各リモートノードはこの要求を受け取り、オペレーションを実行し、その後に応答を返します。
3. その後、ローカルノードが、ローカル応答とリモート応答を収集し、オペレーション自体に必要な削減を実行します。
4. その後、最終的な縮小応答がユーザーに返されます

ほとんどの場合、オペレーションはすべて各リモートノードに完全に適用されるため、すべてのオペレーションは完全に分散されます。通常、複数のノードからの結果を減らすために、最後のオペレーションまたは関連するものだけが再適用される場合があります。重要な点の1つは、実際にはシリアライズする必要がないことに注意してください。これは、希望の部分であるものが最後に送信された最後の値になります (さまざまなオペレーションの例外は以下に強調表示されます)。

**端末オペレーターの分散結果の縮小** 以下の段落では各種の端末オペレーターの分散処理方法を説明します。これらのいくつかは、最終結果の代わりに中間値をシリアル化可能にする必要があるという点で特別です。

### `allMatch` `noneMatch` `anyMatch`

`allMatch` オペレーションは各ノードで実行され、すべての結果が論理的に結合されて適切な値を取得します。`noneMatch` オペレーションおよび `anyMatch` オペレーションは、論理的または代わりに使用します。これらのメソッドは早期終了もサポートしており、最終結果が判明するとリモート操作とローカル操作を停止します。

### `collect`

`collect` メソッドは、いくつかの追加手順を実行できるという点で興味深いものです。リモートノードは、結果に対して最終 `finisher` を実行せず、代わりに完全に結合された結果を送り返すことを除いて、すべてを通常どおり実行します。次に、ローカルスレッドは、リモートとローカルの結果を値



に **結合** し、最終的に終了します。ここで覚えておくべき重要な点は、最終的な値はシリアル化可能である必要はなく、**supplier** メソッドおよび **combiner** メソッドから生成された値である必要があるということです。

#### count

**count** メソッドは、各ノードから番号を一緒に追加します。

#### findAny findFirst

**findAny** オペレーションは、最初に見つかった値 (リモートノードからのものかローカル) を返します。これは、値が見つかるか他の値を処理しないという点で、早期終了をサポートすることに注意してください。**findFirst** メソッドは、ソートされた中間操作が必要になる点で特殊です。これは、**例外** セクションで説明されています。

#### max min

**max** メソッドおよび **min** メソッドは、各ノードの各最小値または最大値を見つけ、最終的にノード間の最小値または最大値のみが返されるようにローカルで実行されます。

#### reduce

さまざまな **reduce** メソッド **1**、**2**、**3** は、アキュムレーターが実行可能な量の結果のシリアライズを最終的に行います。次に、ローカルとリモートの結果をローカルでまとめて累積してから、指定した場合は組み合わせます。これは、組み合わせた値がシリアライズ可能である必要がないことを意味する点に注意してください。

## 13.11. キーベースの再ハッシュ対応 OPERATOR

**iterator**、**spliterator**、および **forEach** は、再ハッシュ認識が、セグメントだけでなくセグメントごとに処理されたキーを追跡する必要がある点で、他のターミナル operator とは異なります。これは、クラスターメンバーシップが変更された場合でも、1回だけ (**iterator** と **spliterator**) または1回以上の (**forEach**) の動作を保証するためです。

リモートノードで呼び出されると **iterator** および **spliterator** オペレーターは、エントリーの再バッチを返します。この場合、次のバッチは最後に使用された後にのみ送信されます。このバッチ処理は、ある時点のメモリー内のエントリー数を制限するために行われます。ユーザーノードは、処理したキーを保持し、特定のセグメントが完了すると、それらのキーをメモリーから解放します。そのため、**iterator** メソッドには順次処理が優先されることがあるため、すべてのノードからではなく、セグメントキーのサブセットのみがメモリーに保持されます。

**forEach()** メソッドはバッチを返しますが、キーの処理が少なくともバッチ処理された後に、キーのバッチを返します。これにより、送信元ノードはどの鍵がすでに処理されているかを把握して、同じエントリーを再処理する可能性を減らすことができます。ただし、これはノードが予期せずダウンした場合に、少なくとも1回の動作を要する可能性があることを意味します。この場合、そのノードはバッチを処理してまだ完了していない可能性があり、処理されたが完了したバッチに含まれていないエントリーは、再ハッシュ失敗オペレーションが発生したときに再度実行されます。ノードを追加しても、すべての応答を受け取るまで、再ハッシュフェイルオーバーが発生しないため、この問題は発生しません。

これらのオペレーションのバッチサイズは両方とも、**CacheStream** で **distributedBatchSize** メソッドを呼び出して設定できる値と同じ値で制御されます。この値はデフォルトで、状態遷移で設定された **chunkSize** に設定されます。残念ながら、この値は、メモリー使用量とパフォーマンスと少なくとも1回のトレードオフであり、マイルージは異なる場合があります。

#### レプリケートされた分散キャッシュでの iterator の使用

ノードが分散ストリームに要求されたすべてのセグメントのプライマリーまたはバックアップ所有者である場合、Data Grid は **iterator** または **spliterator** の端末操作をローカルで実行します。これにより、リモートの反復がリソース集約型であるためにパフォーマンスが最適化されます。

この最適化は、レプリケートされたキャッシュと分散キャッシュの両方に適用されます。ただし、Data Grid は、**shared** および **write-behind** の両方が有効なキャッシュストアを使用する場合にリモートで反復を実行します。この場合は、リモートで反復を行うことで一貫性が確保されます。

## 13.12. 中間オペレーションの例外

特別な例外を持つ中間オペレーションがあります。これらは、`skip`、`peek`、ソートされた `12` および `distinct` です。これらの方法はすべて、正確さを保証するためにストリーム処理に埋め込まれたある種の人為的な iterator を備えています。これらは以下のように文書化されています。このオペレーションにより、パフォーマンスが低下する可能性があります。

### スキップ

中間スキップオペレーションまで人為的な iterator が埋め込まれています。結果はローカルに格納され、適切な要素量をスキップできます。

### ソート済み

警告:この操作には、ローカルノード上のメモリのすべてのエントリが必要です。人為的な iterator は、中間のソートされたオペレーションまで埋め込まれます。すべての結果がローカルでソートされます。要素のバッチを返す分散ソートを計画することは可能ですが、これはまだ実装されていません。

### 一意

警告:この操作には、ローカルノード上のメモリのすべて、またはほぼすべてのエントリが必要です。各リモートノードで `distinct` が実行され、人為的な iterator がそれらの `distinct` 値を返します。そして最後に、これらの結果はすべて、個別のオペレーションが実行されます。

残りの中間オペレーションは、期待通りに完全に配布されます。

## 13.13. 例

### 単語数

単語数は使いすぎると、`map/reduc` パラダイムの典型的な例になります。Data Grid ノードに `key → sentence` が保存されていると仮定します。キーは文字列であり、各文も文字列であり、使用可能なすべての文のすべての単語の出現をカウントする必要があります。このような分散タスクの実装は、以下のように定義できます。

```
public class WordCountExample {
    /**
     * In this example replace c1 and c2 with
     * real Cache references
     *
     * @param args
     */
    public static void main(String[] args) {
        Cache<String, String> c1 = ...;
        Cache<String, String> c2 = ...;

        c1.put("1", "Hello world here I am");
        c2.put("2", "Infinispan rules the world");
        c1.put("3", "JUDCon is in Boston");
        c2.put("4", "JBoss World is in Boston as well");
        c1.put("12", "JBoss Application Server");
        c2.put("15", "Hello world");
    }
}
```

```

c1.put("14", "Infinispan community");
c2.put("15", "Hello world");

c1.put("111", "Infinispan open source");
c2.put("112", "Boston is close to Toronto");
c1.put("113", "Toronto is a capital of Ontario");
c2.put("114", "JUDCon is cool");
c1.put("211", "JBoss World is awesome");
c2.put("212", "JBoss rules");
c1.put("213", "JBoss division of RedHat ");
c2.put("214", "RedHat community");

Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
    .map(e -> e.getValue().split("\\s"))
    .flatMap(Arrays::stream)
    .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
}
}

```

この場合、前述の例から単語数を簡単に実行できます。

ただし、例で最も頻繁に使用される単語を見つけたい場合はどうすればよいでしょうか。このケースについて少し考えてみると、最初にすべての単語をカウントしてローカルで利用できるようにする必要があります。ことに気付くでしょう。そのため、実際にはいくつかのオプションがあります。

コレクターでフィニッシャーを使用できます。これは、すべての結果が収集された後にユーザースレッドで呼び出されます。前の例からいくつかの冗長な行が削除されました。

```

public class WordCountExample {
    public static void main(String[] args) {
        // Lines removed

        String mostFrequentWord = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.collectingAndThen(
                Collectors.groupingBy(Function.identity(), Collectors.counting()),
                wordCountMap -> {
                    String mostFrequent = null;
                    long maxCount = 0;
                    for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
                        int count = e.getValue().intValue();
                        if (count > maxCount) {
                            maxCount = count;
                            mostFrequent = e.getKey();
                        }
                    }
                    return mostFrequent;
                }
            ));
    }
}

```

残念ながら、最後のステップは単一のスレッドでのみ実行されるため、単語が多い場合は非常に遅くなる可能性があります。これを Streams で並列化するもう 1 つの方法があります。



前述したように、処理後にローカルノードに含まれるため、実際にはマップ結果でストリームを使用することができました。そのため、結果に並列ストリームを使用できます。

```
public class WordFrequencyExample {
    public static void main(String[] args) {
        // Lines removed

        Map<String, Long> wordCount = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
        Optional<Map.Entry<String, Long>> mostFrequent =
            wordCount.entrySet().parallelStream().reduce(
                (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
    }
}
```

これにより、最も頻繁に発生する要素を計算する際に、すべてのコアをローカルで利用できるようになります。

### 特定のエントリーの削除

分散ストリームは、ライブ先のデータを変更する方法として使用することもできます。たとえば、特定の単語が含まれるキャッシュのエントリーをすべて削除します。

```
public class RemoveBadWords {
    public static void main(String[] args) {
        // Lines removed
        String word = ..

        c1.entrySet().parallelStream()
            .filter(e -> e.getValue().contains(word))
            .forEach((c, e) -> c.remove(e.getKey()));
    }
}
```

シリアル化されているものとそうでないものを注意深く記録すると、ラムダによって取得されるときに、オペレーションとともに単語のみが他のノードにシリアル化されることがわかります。ただし、実際に節約できるのは、キャッシュ操作がプライマリ所有者に対して実行されるため、これらの値をキャッシュから削除するために必要なネットワークトラフィックの量が削減されることです。各ノードで呼び出されたときにキャッシュを `BiConsumer` に渡す特別な `BiConsumer` メソッドのオーバーライドを提供するため、キャッシュはラムダによって取得されません。

この方法で `forEach` コマンドを使用する際に留意すべきことの1つは、基になるストリームがロックを取得しないことです。キャッシュの削除操作は自然にロックを取得しますが、値はストリームが見たものから変更されている可能性があります。つまり、ストリームがエントリーを読み取った後にエントリーが変更された可能性があります。削除によって実際に削除されました。

`LockedStream` と呼ばれる新しいバリエーションを具体的に追加しました。

### 他の多くの例

`Streams` API は JRE ツールであり、それを使用するための例がたくさんあります。操作は何らかの方法でシリアル化可能である必要があることを覚えておいてください。

## 第14章 JCache (JSR-107) API

Data Grid は JCache 1.0 API ([JSR-107](#)) の実装を提供します。JCache は、一時 Java オブジェクトをメモリーにキャッシュするための標準 Java API を指定します。Java オブジェクトをキャッシュすると、取得にコストがかかるデータ (DB や Web サービスなど) や計算が難しいデータを使用することで発生するボトルネックを回避するのに役立ちます。これらのタイプのオブジェクトをメモリーにキャッシュすると、コストのかかるラウンドトリップや再計算を行う代わりに、メモリーから直接データを取得することで、アプリケーションのパフォーマンスを高速化できます。本書では、仕様の Data Grid 実装で JCache を使用方法と、API の主要な側面が説明されています。

### 14.1. 組み込みキャッシュの作成

#### 前提条件

1. `cache-api` がクラスパスにあることを確認します。
2. 以下の依存関係を `pom.xml` に追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
</dependency>
```

#### 手順

- 以下のように、デフォルトの JCache API 設定を使用する組み込みキャッシュを作成します。

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```

#### 14.1.1. 組み込みキャッシュの設定

- 以下のように、カスタム Data Grid 設定の URI を `CachingProvider.getCacheManager(URI)` 呼び出しに渡します。

```
import java.net.URI;
import javax.cache.*;
import javax.cache.configuration.*;

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/infinispan.xml");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("infinispan.xml").toURI();

// Create a cache manager using the above configuration
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager(uri,
    this.getClass().getClassLoader(), null);
```



### 警告

デフォルトでは、JCache API はデータを **storeByValue** として保存するように指定しているため、キャッシュへの操作以外のオブジェクト状態の変更は、キャッシュに保存されているオブジェクトに影響を与えません。Data Grid はこれまで、シリアル化/マーシャリングを使用してこれを実装し、コピーを作成してキャッシュに保存しており、その方法は仕様に準拠しています。したがって、Data Grid でデフォルトの JCache 設定を使用する場合、保存されるデータはマーシャリング可能である必要があります。

または、(Data Grid または JDK Collections が機能するのと同じように) 参照によってデータを格納するように JCache を設定することもできます。これを行うには、次のコマンドを実行します。

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>().setStoreByValue(false));
```

## 14.2. リモートキャッシュの作成

### 前提条件

1. **cache-api** がクラスパスにあることを確認します。
2. 以下の依存関係を **pom.xml** に追加します。

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache-remote</artifactId>
</dependency>
```

### 手順

- リモート Data Grid サーバーでキャッシュを作成し、以下のようにデフォルトの JCache API 設定を使用します。

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager via org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager =
    Caching.getCachingProvider("org.infinispan.jcache.remote.JCachingProvider").getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("remoteNamedCache",
    new MutableConfiguration<String, String>());
```

### 14.2.1. リモートキャッシュの設定

Hot Rod 設定ファイルには、リモートキャッシュのカスタマイズに使用できる **infinispan.client.hotrod.cache.\*** プロパティーが含まれます。

- 以下のように、**hotrod-client.properties** ファイルの URI を **CachingProvider.getCacheManager(URI)** 呼び出しに渡します。

```
import javax.cache.*;
import javax.cache.configuration.*;

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/hotrod-client.properties");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("hotrod-client.properties").toURI();

// Retrieve the system wide cache manager via org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager =
    Caching.getCacheManager("org.infinispan.jcache.remote.JCachingProvider")
        .getCacheManager(uri, this.getClass().getClassLoader(), null);
```

### 14.3. データの保管および取得

JCache の API が `java.util.Map` または `java.util.concurrent.ConcurrentMap` のいずれも拡張していないにもかかわらず、キー/値の API を提供してデータを格納および取得します。

```
import javax.cache.*;
import javax.cache.configuration.*;

CacheManager cacheManager = Caching.getCacheManager().getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

標準の `java.util.Map` とは異なり、`javax.cache.Cache` には `put` と `getAndPut` と呼ばれる 2 つの基本的な `put` メソッドが含まれています。前者は `void` を返しますが、後者はキーに関連付けられた以前の値を返します。そのため、JCache の `java.util.Map.put(K)` に相当するものは `javax.cache.Cache.getAndPut(K)` になります。

#### ヒント

JCache API はスタンドアロンキャッシングのみを対象としていますが、永続ストアにプラグインすることができ、クラスタリングまたは分散を念頭に置いて設計されています。`javax.cache.Cache` が 2 つの `put` メソッドを提供する理由は、標準の `java.util.Map.put` 呼び出しにより以前の値を計算するためです。永続ストアが使用されている場合、またはキャッシュが分散されている場合、前の値を返すことはコストのかかる操作になる可能性があり、多くの場合、ユーザーは戻り値を使用せずに標準の `java.util.Map.put(K)` を呼び出します。そのため、JCache ユーザーは戻り値が関連するかどうかについて考慮する必要があります。この場合、`javax.cache.Cache.getAndPut(K)` を呼び出す必要があります。それ以外の場合は、`java.util.Map.put(K, V)` を呼び出すことができ、以前の値を返すようなコストのかかる操作が返されなくなります。

### 14.4. JAVA.UTIL.CONCURRENT.CONCURRENTMAP と JAVAX.CACHE.CACHE APIS の比較

ここでは、`java.util.concurrent.ConcurrentMap` および `javax.cache.Cache` API によって提供されるデータ操作 API を簡単に比較します。

操作	java.util.concurrent.Concurrent Map<K, V>	javax.cache.Cache<K, V>
保存して返さない	該当なし	<b>void put(K key)</b>
保存して以前の値を返す	<b>V put(K key)</b>	<b>V getAndPut(K key)</b>
存在しない場合は保存する	<b>V putIfAbsent(K key, V value)</b>	<b>boolean putIfAbsent(K key, V value)</b>
取得	<b>V get(Object key)</b>	<b>V get(K key)</b>
存在する場合は削除	<b>V remove(Object key)</b>	<b>boolean remove(K key)</b>
以前の値を削除して返す	<b>V remove(Object key)</b>	<b>V getAndRemove(K key)</b>
条件の削除	<b>boolean remove(Object key, Object value)</b>	<b>boolean remove(K key, V oldValue)</b>
存在する場合は置き換え	<b>V replace(K key, V value)</b>	<b>boolean replace(K key, V value)</b>
以前の値を置き換えて返す	<b>V replace(K key, V value)</b>	<b>V getAndReplace(K key, V value)</b>
条件の置き換え	<b>boolean replace(K key, V oldValue, V newValue)</b>	<b>boolean replace(K key, V oldValue, V newValue)</b>

2つのAPIを比較すると、可能であれば、JCacheが以前の値を返さないようにして、コストのかかるネットワークまたはIO操作を実行するオペレーションを回避していることがわかります。これは、JCache APIの設計における最も重要な原則です。実際、[java.util.concurrent.ConcurrentMap](#)には存在するが、分散キャッシュでの計算にコストがかかる可能性があるため、[javax.cache.Cache](#)には存在しない一連のオペレーションがあります。唯一の例外は、キャッシュの内容を反復処理することです。

操作	java.util.concurrent.Concurrent Map<K, V>	javax.cache.Cache<K, V>
キャッシュのサイズを計算する	<b>int size()</b>	該当なし
キャッシュのすべてのキーを返す	<b>Set&lt;K&gt; keySet()</b>	該当なし
キャッシュのすべての値を返す	<b>Collection&lt;V&gt; values()</b>	該当なし
キャッシュ内のすべてのエントリを返す	<b>Set&lt;Map.Entry&lt;K, V&gt;&gt; entrySet()</b>	該当なし

操作	java.util.concurrent.Concurrent Map<K, V>	javax.cache.Cache<K, V>
キャッシュを繰り返し処理する	keySet、value、または entrySet で <b>iterator()</b> メソッドを使用します。	<b>Iterator&lt;Cache.Entry&lt;K, V&gt;&gt; iterator()</b>

## 14.5. JCACHE インスタンスのクラスタリング

Data Grid JCache 実装は仕様を越え、標準 API を使用してクラスターキャッシュを使用できるようになります。次のようにキャッシュを複製するように設定された Data Grid 設定ファイルがあるとします。

### infinispan.xml

```
<infinispan>
  <cache-container default-cache="namedCache">
    <transport cluster="jcache-cluster" />
    <replicated-cache name="namedCache" />
  </cache-container>
</infinispan>
```

このコードを使用して、キャッシュのクラスターを作成できます。

```
import javax.cache.*;
import java.net.URI;

// For multiple cache managers to be constructed with the standard JCache API
// and live in the same JVM, either their names, or their classloaders, must
// be different.
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is working

// --

public static class TestClassLoader extends ClassLoader {
    public TestClassLoader(ClassLoader parent) {
        super(parent);
    }
}
```

## 第15章 マルチマップキャッシュ

MultimapCache は、各キーに複数の値を含めることができる値にキーをマップする Data Grid キャッシュの一種です。

### 15.1. インストールと設定

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
</dependency>
```

### 15.2. MULTIMAPCACHE API

MultimapCache API は、Multimap キャッシュと対話する複数のメソッドを公開します。これらのメソッドは、ほとんどの場合、ノンブロッキングです。詳細については、[制限](#)を参照してください。

```
public interface MultimapCache<K, V> {

  CompletableFuture<Optional<CacheEntry<K, Collection<V>>>> getEntry(K key);

  CompletableFuture<Void> remove(SerializablePredicate<? super V> p);

  CompletableFuture<Void> put(K key, V value);

  CompletableFuture<Collection<V>> get(K key);

  CompletableFuture<Boolean> remove(K key);

  CompletableFuture<Boolean> remove(K key, V value);

  CompletableFuture<Void> remove(Predicate<? super V> p);

  CompletableFuture<Boolean> containsKey(K key);

  CompletableFuture<Boolean> containsValue(V value);

  CompletableFuture<Boolean> containsEntry(K key, V value);

  CompletableFuture<Long> size();

  boolean supportsDuplicates();

}
```

#### CompletableFuture<Void> put(K key, V value)

キーと値のペアをマルチマップキャッシュに配置します。

```
MultimapCache<String, String> multimapCache = ...;
```



```

multimapCache.put("girlNames", "marie")
    .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
    .thenCompose(r3 -> multimapCache.get("girlNames"))
    .thenAccept(names -> {
        if(names.contains("marie"))
            System.out.println("Marie is a girl name");

        if(names.contains("oihana"))
            System.out.println("Oihana is a girl name");
    });

```

このコードの出力は以下のようになります。

```

Marie is a girl name
Oihana is a girl name

```

### CompletableFuture<Collection<V>> get(K key)

存在する場合、このマルチマップキャッシュ内のキーに関連付けられた値のビューコレクションを返す非同期。取得したコレクションへの変更は、このマルチマップキャッシュの値を変更しません。このメソッドは空のコレクションを返すと、キーが見つからないことを意味します。

### CompletableFuture<Boolean> remove(K key)

キーに関連付けられたエントリがマルチマップキャッシュに存在する場合は、それを削除する非同期。

### CompletableFuture<Boolean> remove(K key, V value)

キーと値のペアが存在する場合は、マルチマップキャッシュから削除する非同期。

### CompletableFuture<Void> remove(Predicate<? super V> p)

非同期メソッド。指定の述語に一致するすべての値を削除します。

### CompletableFuture<Boolean> containsKey(K key)

このマルチマップにキーが含まれる場合に true を返す非同期。

### CompletableFuture<Boolean> containsValue(V value)

このマルチマップに少なくとも1つのキーの値が含まれている場合に true を返す非同期。

### CompletableFuture<Boolean> containsEntry(K key, V value)

このマルチマップに値を持つキーと値のペアが1つ以上含まれている場合、true を返す非同期。

### CompletableFuture<Long> size()

マルチマップキャッシュ内のキーと値のペアの数を返す非同期。明確な数のキーは返されません。

### boolean supportsDuplicates()

マルチマップキャッシュが重複をサポートする場合は true を返す非同期。これは、マルチマップのコンテンツが 'a' → ['1', '1', '2'] になる可能性があることを意味します。重複はまだサポートされていないため、今のところ、このメソッドは常に false を返します。指定された値の存在は、'equals' および 'hashCode' method' のコントラクトによって決定されます。



## 15.3. マルチマップキャッシュの作成

現在、MultimapCache は通常のキャッシュとして設定されます。これは、コードまたは XML 設定のいずれかで実行できます。[configure a cache]へのセクションリンクで、通常のキャッシュを設定する方法を参照してください。

### 15.3.1. 組み込みモード

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager cm = ... ;

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager =
EmbeddedMultimapCacheManagerFactory.from(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

## 15.4. 制限事項

ほとんどの場合、Multimap キャッシュは通常のキャッシュとして動作しますが、以下のように現在のバージョンにはいくつかの制限があります。

### 15.4.1. 重複のサポート

重複はまだサポートされていません。これは、マルチマップに重複したキーと値のペアが含まれていないことを意味します。put メソッドが呼び出されるたびに、キーと値のペアがすでに存在する場合、このキーと値のペアは追加されません。Multimap にキーと値のペアがすでに存在しているかどうかを確認するために使用されるメソッドは **equals** および **hashCode** です。

### 15.4.2. エビクション

現時点では、エビクションはキーと値のペアごとではなく、キーごとに機能します。これは、キーがエビクトされるたびに、キーに関連付けられているすべての値も削除されることを意味します。

### 15.4.3. トランザクション

暗黙的なトランザクションは、自動コミットによってサポートされ、すべてのメソッドは非ブロッキングです。ほとんどの場合、明示的なトランザクションはブロックせずに機能します。ブロックするメソッドは **size**、**containsEntry**、および **remove(Predicate<? super V> p)** です。

## 第16章 RED HAT JBOSS EAP のデータグリッドモジュール

Red Hat JBoss EAP にデプロイされたアプリケーション内で Data Grid を使用するには、以下を実行する Data Grid モジュールをインストールする必要があります。

- WAR または EAR ファイルに Data Grid JAR ファイルをパッケージ化せずにアプリケーションをデプロイできます。
- Red Hat JBoss EAP にバンドルされているバージョンとは独立した Data Grid を使用できるようにします。



### 重要

Data Grid モジュールは非推奨となり、削除される予定です。これらのモジュールは、Red Hat JBoss EAP が **infinispan** サブシステムを直接管理するまで一時的なソリューションを提供します。

### 16.1. DATA GRID モジュールのインストール

Red Hat JBoss EAP の Data Grid モジュールをダウンロードしてインストールします。

#### 前提条件

1. JDK 8 以降。
2. 既存の Red Hat JBoss EAP インストール

#### 手順

1. Red Hat カスタマーポータルにログインします。
2. [Data Grid ソフトウェアダウンロード](#) からモジュールの ZIP アーカイブをダウンロードします。
3. ZIP アーカイブを抽出し、**modules** の内容を Red Hat JBoss EAP インストールの **modules** ディレクトリーにコピーして、結果の構造を取得します。  
**\$EAP\_HOME/modules/system/add-ons/rhdg/org/infinispan/rhdg-8.4**

### 16.2. DATA GRID モジュールを使用するためのアプリケーションの設定

Red Hat JBoss EAP 用の DataGrid モジュールをインストールした後、Data Grid 機能を使用するようにアプリケーションを設定します。

#### 手順

1. プロジェクトの **pom.xml** ファイルで、必要な Data Grid の依存関係を **提供されているもの**としてマークします。
2. 適切な **MANIFEST.MF** ファイルを生成するようにアーティファクトアーカイバを設定します。

#### pom.xml

```
<dependencies>  
<dependency>
```

```
<groupId>org.infinispan</groupId>
<artifactId>infinispan-core</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cache-store-jdbc</artifactId>
  <scope>provided</scope>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <archive>
          <manifestEntries>
            <Dependencies>org.infinispan:rhdg-8.1 services</Dependencies>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Data Grid 機能は、以下のようにアプリケーションのマニフェストにエントリーとして追加できる単一のモジュール **org.infinispan** としてパッケージ化されます。

## MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:rhdg-8.1 services
```

## 第17章 RED HAT JBOSS WEB SERVER から RED HAT DATA GRID への HTTP セッションの外部的化

**org.apache.catalina.Manager** インターフェイスを使用して、HTTP セッションデータを JBoss Web Server デプロイメントから Data Grid Server クラスターに外部化して、高可用性を実現します。

### 17.1. TOMCAT セッションクライアントのインストール

Tomcat セッションクライアントをインストールし、Red Hat JBoss Web Server アプリケーションから Red Hat Data Grid に HTTP セッションを外部的化します。

#### 手順

1. [Data Grid Software Downloads](#) から **redhat-datagrid-8.1.1-tomcat<\$version>-session-client.zip** アーカイブをダウンロードします。
2. アーカイブをローカルのファイルシステムにデプロイメントします。
3. デプロイメントしたアーカイブから **\$CATALINA\_HOME/lib** に **lib/** ディレクトリーの内容をコピーします。

### 17.2. セッションマネージャーの設定

セッションマネージャーの **HotRodManager** クラスを設定し、Tomcat セッションクライアントが Red Hat Data Grid Server に接続し、データをリモートキャッシュに保存する方法を定義します。

#### 前提条件

- Tomcat セッションクライアントをインストールします。
- 1つ以上の Data Grid Server インスタンスをインストールします。
- HTTP セッションデータを保存するテンプレートとして使用する Data Grid Server にキャッシュを作成します。

#### 手順

1. **\$CATALINA\_HOME/conf/context.xml** または **/WEB-INF/context.xml** を開いて編集します。
2. **org.wildfly.clustering.tomcat.hotrod.HotRodManager** を **className** プロパティーの値として指定します。
3. **configurationName** プロパティーで、テンプレートとして使用するキャッシュの名前を指定します。
4. **HotRodManager** クラスの他の設定プロパティーを必要に応じて定義します。
5. **infinispan.client.hotrod**. 接頭辞なしで Hot Rod クライアント設定プロパティーを設定します。
  - a. **server\_list** プロパティーで Data Grid Server ノードのリストを指定します。
  - b. **auth\_username** および **auth\_password** プロパティーで Data Grid のクレデンシャルを指定します。

6. 必要に応じて、Tomcat セッションマネージャーの一般的な属性を指定します。
7. **context.xml** を保存して閉じます。

## 設定例

```
<Manager className="org.wildfly.clustering.tomcat.hotrod.HotRodManager"
  configurationName="mycache"
  persistenceStrategy="FINE"
  maxActiveSessions="100"
  server_list="192.0.2.0:11222;192.0.2.0:11223;192.0.2.0:11224"
  protocol_version="2.9"
  auth_username="admin"
  auth_password="changeme"
  auth_realm="default"
  sasl_mechanism="DIGEST-MD5"
  auth_server_name="infinispan"/>
```

## 検証

Tomcat セッションクライアントがリモートキャッシュにデータを保存することを確認するには、以下を行います。

1. 任意のブラウザで Data Grid コンソールを開きます。  
デフォルトでは、コンソールは **http://127.0.0.1:11222/console/** で利用できます。
2. Tomcat セッションクライアントが、デプロイされたアプリケーションごとにキャッシュを作成することを確認します。

### 17.2.1. Hot Rod マネージャー設定プロパティ

以下の表は、**HotRodManager** クラスの設定プロパティをリストおよび説明しています。

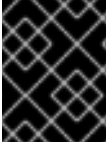
プロパティ	説明
<b>className</b>	<b>org.wildfly.clustering.tomcat.hotrod.HotRodManager</b> をセッションマネージャーとして指定します。
<b>configurationName</b>	HTTP セッションデータを保存するテンプレートとして使用する Data Grid Server のリモートキャッシュを指定します。
<b>persistenceStrategy</b>	セッションをキャッシュのエントリにマップする方法を定義します。  <b>COARSE</b> は、単一のキャッシュエントリ内にすべてのセッション属性を保存します。これはデフォルトになります。  <b>FINE</b> は、セッション属性を個別のキャッシュエントリに保存します。

プロパティ	説明
<b>maxActiveSessions</b>	キャッシュに保存するセッションの最大数を定義します。デフォルトは最大値なし (制限なし) です。

#### 関連情報

- [Data Grid のドキュメント](#)
- [Tomcat 7.0 Manager の共通属性](#)
- [Tomcat 8.0 Manager の共通属性](#)
- [Tomcat 8.5 Manager の共通属性](#)
- [Tomcat 9.0 Manager の共通属性](#)

## 第18章 カスタムインターセプター



### 重要

カスタムインターセプターは Data Grid で非推奨となり、今後のバージョンで削除されます。

カスタムインターセプターは、キャッシュへの変更に影響を与えたり、それに応答したりできるようにすることで、Data Grid を拡張する方法です。要素が追加/削除/更新されること、またはトランザクションがコミットされることが、このような変更の例としてあげられます。

### 18.1. カスタムインターセプターの宣言的追加

カスタムのインターセプターは、名前付きキャッシュごとに追加できます。これは、名前の付いた各キャッシュに独自のインターセプタースタックがあるためです。以下の xml スニペットは、カスタムインターセプターを追加する方法を示しています。

```
<local-cache name="cacheWithCustomInterceptors">
  <!--
    Define custom interceptors. All custom interceptors need to extend
    org.jboss.cache.interceptors.base.CommandInterceptor
  -->
  <custom-interceptors>
    <interceptor position="FIRST" class="com.mycompany.CustomInterceptor1">
      <property name="attributeOne">value1</property>
      <property name="attributeTwo">value2</property>
    </interceptor>
    <interceptor position="LAST" class="com.mycompany.CustomInterceptor2"/>
    <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
    <interceptor before="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor2"/>
    <interceptor after="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor1"/>
  </custom-interceptors>
</local-cache>
```

### 18.2. プログラムによるカスタムインターセプターの追加

そのためには、[AdvancedCache](#) への参照を取得する必要があります。これは、以下のように実行できます。

```
CacheManager cm = getCacheManager();//magic
Cache aCache = cm.getCache("aName");
AdvancedCache advCache = aCache.getAdvancedCache();
```

次に、`addInterceptor()` メソッドの1つを使用して、実際のインターセプターを追加する必要があります。詳細は、[AdvancedCache](#) javadoc を参照してください。

### 18.3. カスタムインターセプターの設計

カスタムインターセプターを作成するときは、次のルールに従う必要があります。

- カスタムインターセプターは、構築を有効にするために、パブリックの空のコンストラクターを宣言する必要があります。
- カスタムインターセプターには、XML 設定で使用されるプロパティータグで定義されたすべてのプロパティに対するセッターがあります。